The VTune™ Performance Analyzer Reader/Writer API (TBRW)

User Guide

Document Number: 320237-002US

Revision: 2.3

World Wide Web: http://www.intel.com

Disclaimer and Legal Information

Revision History

Document Number	Revision Number	Description	Revision Date
320237-001	2.2	Initial release.	July, 2008
320237-002	2.3	Changed the code example, added information on the VTune™ Performance Analyzer compatible files.	May, 2009

Contents

1 About this Document

This VTune Performance Analyzer™ reader/writer API (TBRW) enables developers to read and write persisted data in an on-disk format that is compatible with the VTune™ Performance Analyzer. This API supports tb5 files, version 16 and higher.

1.1 Intended Audience

Read this document if you are interested in reading and writing persisted data in a VTune analyzer compatible on-disk format.

1.2 Contents of the TBRW Package

The TBRW package contains the following list of directories.

The VTune analyzer TBRW (Tb5 Read Write) related files are located in the Intel VTune analyzer installation directory structure as shown below.

```
+ analyzer
   + bin          (TBRW binaries tbrw.dll/libtbrw.so or
   |              sampling_utils.dll/libsampling_utils.so)
  + include
      + samprec_shared.h  (TBRW header with data description enumerations)
      + tbrw.h            (TBRW header with TBRW API)
      + tbrw_types.h      (TBRW header wiht TBRW data types)
   + lib      (TBRW libraries tbrw.lib and sampling_utils.lib on Windows)
+ examples                (example programs)
   + TBRW         (TBRW examples)
      + TBRWExamples.sln  (Microsoft* Visual Studio* 2005 solution file to
      |                   build TBRW examples on Windows* OS)
      + linux_setenv      environment for building examples on Linux* OS)
      + Makefile          (makefile for building examples on Linux)
      + print_tb5.cpp     (example program to print tb5 content)
      + tbrw_reader.cpp   (example program to read tb5 and generate data for
      |                   tbrw_writer program)
      + tbrw_writer.cpp   (example program to simulate creation of tb5 data
      |                   file using TBRW API, uses tbrw_reader data)
      + tbrw_print    (Visual Studio project file for building tbrw_print)
      + tbrw_reader   (Visual Studio project file for building tbrw_reader)
      + tbrw_writer    (Visual Studio project file for building tbrw_writer)
      + sample.tb5     (A sample tb5 data file)
+ doc
   + TBRW.pdf              (TBRW API documentation)
```

1.2.1 TBRW Binaries

The bin folder in the TBRW package contains the TBRW binaries for respective platform/architecture combination.

You need to link to these binaries to use the TBRW API.

Table 1 TBRW Binary Files

Linux* OS	libtbrw.so - contains TBRW API (IA-32) libsampling_utils.so - contains utility API used by TBRW
Window* OS	tbrw.lib and tbrw.dll - TBRW binary dll, contains TBRW API sampling_utils.lib and sampling_utils.dll - Contains utility API used by TBRW

1.2.2 TBRW Headers

The include folder in the VTune analyzer installation contains the header files required to use TBRW API.

You need to include `tbrw.h` and `samprec_shared.h`.

The `tbrw.h` header file contains the API declarations exposed by TBRW and the TBRW data structures that are used in the TBRW API interface.

The `samprec_shared.h` contains the list of data descriptor types that can be used in writing the data section of a tb5 file. See 6Writing VTune™ Performance Analyzer Compatible Files for more information.

1.3 Goals

This document enables you to do the following:

- Provide sampling data to the VTune analyzer in a way that the analyzer can interpret it.
- Define the sample record to best meet your needs (constrained by (1) above)
- Define the size of a sample record on a per-data stream basis.
- Add data to a current “section”. However, once a section is “done” no more data can be added to that section.

1.3.1 API Standard

- The smallest granularity for defining a data field in a sample record is one byte.
- Any file that is written via this API is validated in some form to minimize the possibility of accidentally persisting data that cannot be read or manipulated by the VTune analyzer.
- All strings are Unicode strings.

1.4 Conventions and Symbols

The following conventions are used in this document.

Table 2 Conventions and Symbols used in this Document

`This type style`	Indicates an element of syntax, reserved word, keyword, filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
This type style	Indicates the exact characters you type as input. Also used to highlight the elements of a graphical user interface such as buttons and menu names.
This type style	Indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder.
[items]	Indicates that the items enclosed in brackets are optional.
{ item | item }	Indicates to select only one of the items listed between braces. A vertical bar (|) separates the items.

2 TBRW Examples

The installation contains three sample programs in the `examples/TBRW` folder that demonstrate the TBRW API usage.

2.1 print_tb5

This program reads the data from the provided tb5 file and prints the data for each section of the tb5 file.

Usage

```
print_tb5 <tb5filename>
```

Where `<tb5filename>` is the name of the tb5 file to be read.

2.2 tbrw_reader

This program reads the data from the provided tb5 file and creates two binary data files. This program also creates another binary data file that only contains the sampling data section.

- The first file contains the binary data of each section of the TBRW data file.
- The second file contains the binary data from the data section of the TBRW file. The second file is referred by the first file.

  These two files are used by the `tbrw_writer` program as input to create a tb5 file.

Usage

```
tbrw_reader <unbound tb5 file> <output file name>
```

Where

`<unbound tb5 file>` is the name of the tb5 file from which data needs to be read.

`<output file name>` is the file into which the tb5 data is written.

2.3 tbrw_writer

This program demonstrates the usage of TBRW API to create a tb5 file. Since the example does not perform any real collection of the data, the output from `tbrw_reader` is used as input data. The `tbrw_reader` program can read an existing tb5 file and produce the binary data consumable by `tbrw_writer` program. The `tbrw_writer` program produces a new tb5 file.

Usage

```
tbrw_writer.exe <input_data_file_name> <tb5_file_name>
```

Where

`<input_data_file_name>` is name of the file that contains the sampling binary data.

`<tb5_file_name>` is the output tb5 file that is written with the tb5 sections.

2.4 Building Examples

This section explains how to build examples on Windows and Linux operating systems

2.4.1 Building Examples on Windows* OS

1. Go to `examples\TBRW` folder.
2. Double click `TBRWExamples.sln` to open the examples solution in Visual Studio 2005.

   This workspace contains three projects - `print_tb5`, `tbrw_reader` and `tbrw_writer`.
3. Select the build configuration and build the projects. Make sure that `tbrw.lib` binary and the header file paths are included in the Solution `include/lib` directories.

   The executables for the example binaries are located under `examples/TBRW/<`*`PlatformName`*`>/<Debug|Release>` folder

   Where <*`PlatformName`*> is `Win32` or `x64` or `Itanium`, depending on the architecture.

2.4.2 Building Examples on Linux* OS

Building the examples on a Linux OS requires the following applications and files:

- icc or gcc compiler. To compile with the icc compiler, source the `iccvars.sh` from icc compiler installation and modify the `USE_COMPILER` setting in the Makefile to "icc".
- TBRW header files in `$(VTUNE_HOME)/analyzer/include` directory and the binaries from `$(VTUNE_HOME)/analyzer/bin` directory. On Intel® 64 archtecture, the native TBRW binaries are in: `${VTUNE_HOME}/rdc/analyzer/bin`.The Makefile adds these directories for compilation.

  In addition, there are several conditions that may require specific configurations:

- The VTune analyzer installation is performed under root user, therefore building or running the TBRW examples inside `${VTUNE_HOME}/samples/TBRW` directory requires root user permission. To build/run the examples as a non-root user, copy the `${VTUNE_HOME}/samples/TBRW` directory to a local directory with write permissions.
- If your local gcc library version is not `libstdc++.so.5`, you need to explicitly add local library paths to `LD_LIBRARY_PATH` in the `linux_setenv` file.
  For example, to build and run the examples compiled on IA-32 architecture with gcc 4.x against `libstdc++.so.6`, you need to add the `/lib` directory to `LD_LIBRARY_PATH` in `linux_setenv`, before the path to gcc libaries packaged with the VTune analyzer.

  ```
  export LD_LIBRARY_PATH=/lib:${VTUNE_HOME}/gcc-
  3.3/lib32:${LD_LIBRARY_PATH}:
  ```

  On Intel® 64 architecture:

  ```
  export LD_LIBRARY_PATH=/lib64:${VTUNE_HOME}/gcc-
  3.4/libem64t:${LD_LIBRARY_PATH}:
  ```

- The `VTUNE_HOME` variable points to the default VTune analyzer installation home directory. You need to configure this variable value if the VTune analyzer installation directory is not in the default location.

1. Change directory to `${VTUNE_HOME}/samples/TBRW`.
2. Run the following command to setup the environment.
   ```
   $source linux_setenv
   ```
3. Run the following commands to perform various make operations
   ```
      $make clean     ---- to clean the compiled content
      $make all       ---- to build all examples
      $make debug     ---- to build in debug mode
      $make release   ---- to build in release mode
      $make print_tb5 ---- to build a specific example
   ```

The executables for the examples are in: `${VTUNE_HOME}/samples/TBRW`

To compile and run the examples, you also need the C++ runtime libraries `libimf.so`, `libcxaguard.so.x` and `libstdc++.so.x`. These are packaged with VTune analyzer installation and available in:

`$(VTUNE_HOME)/gcc-3.4/libXXX` (`gcc-3.3` for IA-32).

The `linux_setenv` script adds the appropriate platform specific directories to `LD_LIBRARY_PATH`.

2.5 Running the Examples

A sample tb5 file called `sample.tb5` is provided in the `examples/TBRW` folder the TBRW package.

To run the examples, you need to set the path environment variable on your OS.

On Windows OS:

- Copy the `sample.tb5` file to the `examples/TBRW/<PlatformName>/<Debug|Release>` depending on the configuration.
- Add the `tbrw.dll` file (located in `analyzer\bin` directory) to the PATH environment variable.

On Linux OS:

- Add the path to the `libtbrw.so` file (located in `analyzer/bin` directory) to `LD_LIBRARY_PATH`
- Add the `libimf.so`, `libcxaguard.so` and `libstdc++.so` binaries to the `LD_LIBRARY_PATH` variable. (sourcing `linux_setenv` file takes care of this)

To print the contents of the tb5 file run the following command. This prints the tb5 section information to the console.

```
$print_tb5 sample.tb5
```

To read the tb5 content and create the binary data files useful for tbrw_writer program, run the following command. This creates two output files: `sample.dat`, `sample.tb5.dat`

```
$tbrw_reader sample.tb5 sample.dat
```

To create a tb5 file from the raw input data files use the tbrw_reader. This uses the `sample.dat` and `sample.tb5.dat` files from tbrw_reader run and creates a new tb5 file `samplenew.tb5`

```
$tbrw_writer sample.dat samplenew.tb5
```

3 Overview

The VTune analyzer reader/writer API uses a model similar to file I/O's `open()`/`close()` where the VTune analyzer reader/writer API open equivalent passes back an opaque data structure, similar to a file descriptor from an `open()` call. The data structure must then be passed to any subsequent VTune analyzer reader/writer API call.

The API includes a function which can be used to translate the VTune analyzer error numbers into human readable text strings.

Although the VTune analyzer reader/writer API may use exception handling within the API itself, it does not detect error conditions outside of the API nor the termination of the process. For example, it does not use signals or `atexit()` routines. Therefore, the API needs to be explicitly informed when to abort and cleanup. An API is provided for that purpose.

After you call the `TBRW_open()` routine, you must call either the `TBRW_close()` routine for normal close, or the `TBRW_abort_cleanup_and_close()` routine for exception or abnormal process termination, before the process terminates. Failure to do so can lead to incomplete or temporary files being left after the program has exited.

The VTune analyzer reader/writer API uses the concept of sections to handle different types of data. Some sections have very specific requirements regarding what is in them while others are more free-form. In addition, there is a concept of "global" sections which contain universal information that is related to the data generation run vs. "local" sections which contain data which may be specific to only a portion of the run. Sections are discussed in more detail later in the document.

3.1 Concepts

This section defines concepts that are used throughout the rest of the document.

3.1.1 Definitions

The following table provides some definitions of basic concepts used in this document:

Table 3 Definitions

Definition	Description
VTune analyzer	The VTune™ Performance Analyzer product

VTune analyzer reader/writer API	An API for reading and writing VTune analyzer files. Any file that is written via this API is validated in some form to minimize the possibility of accidentally persisting data that cannot be read or manipulated by the VTune analyzer. The persisted data can be larger than the available virtual memory. You can provide data to the API's as either pointers to data in memory or "pointers" to disk files.
sampling data	Data collected from a sampling session, also referred to as the data samples. A single sample is called a sample record.
binding	Sampling data needs to be post-processed after collection to make the data useful. The result of binding is that each data sample is associated with a module. For more information, see What Does "binding" Mean?
bound data	Data that has been through binding is called bound data.
unbound data, a.k.a. raw data	Sampling data before it has been bound. That is, the module that triggered the samples collected is not known.
VTune analyzer File, a.k.a. TB5 file	A file created by the VTune analyzer reader/writer API. Each VTune analyzer file contains zero or one hardware sections, zero or one software sections, zero or one process/thread sections, zero or one module information sections, zero or one user-defined global sections, and zero or more numbered data streams (the first data stream is number zero). You can open an existing VTune analyzer file and append a new data stream (such as an aggregation of an existing data stream). However, you cannot modify an existing data stream or any of the other existing sections in the file.

3.2 TB5 File Sections

The TB5 file includes global sections and stream sections. A global section is a section that is not a stream section.

3.2.1 Global Sections

The global sections are briefly described in the following table. For complete information, see API Data Structure Reference.

Table 4 Global Sections

Section	Description	Required information
Hardware section	Describes the hardware at the time of data collection.	Number of nodes, number of processors per node, physical memory per node, per processor architecture, per processor feature information, number of processors per package, per processor cache information, per processor speed, per processor front-side bus speed, and timestamp skew between

		processors.
Software section	Describes the software at the time of data collection.	OS information, hostname, IP address, page size, allocation granularity, minimum application address, and maximum application address.
Process/thread section	Used to map the process/thread information from a sample, back to a specific process or thread. If the process/thread section does not exist, the VTune analyzer can still present data, by creating this section from module section. The VTune analyzer the module section in order to map IP address back to symbols, and eventually back to source code.	Sample count, creation/terminate indication, process id, process name, thread id, and thread name.
Module section	Used to map the module information from a sample back to a binary image. If the module section does not exist, the VTune analyzer can still present data, but only based on IP. The VTune analyzer needs the process/thread section as well as the module section in order to map IP address back to symbols and eventually back to source code.	Sample count, load/unload indication, associated pid, path to module, module name, module load address, and module length.
User-defined global section	This is a free-format section that you can use to store data. The data in this section is never interpreted nor used by the VTune analyzer.	
Version information global section	Contains information about the versions of the libraries used to collect and analyze the data found in the file.	This section contains the sampling driver version, binding library version, sample file format version.

3.2.2 Data Stream Sections

A data stream section contains a set of data corresponding to a particular span of time. Each data stream contains the following elements:

- one stream information section
- zero or one *local user-defined* sections
- zero or one *event description* sections
- one data section, and
- one *data description* section that describes the meaning of the data in the data section.

The first data stream is number zero. Stream zero is reserved for legacy-formatted sample record information. Attempts to record legacy sample records into streams one or above results in an error.

The stream sections are briefly described in the following table. For complete information on the sections, see API Data Structure Reference:

Table 5 Data Stream Sections

Section	Description	Information required
Stream information section	Contains information on the type of data in the stream	A comment that describes the data stream in a human readable form, the type of the data stream, such as, sampling data or aggregated data, sampling duration, start and end times of the sampling session, command line string, and cpu mask.
User-defined stream section	Free format section that you can use to store data. The data in this section is never interpreted nor used by the VTune analyzer.	
Data description section	Contains one *data descriptor* that defines the meaning of the data in a data section. This section must exist for the VTune analyzer to do post-collection analysis. The VTune analyzer uses the data descriptor to determine if certain types of information are available in the data samples. In some cases, when that type of data isn't present in the samples, the VTune analyzer can make a simplifying assumption and things continue to work as expected. For example, if there is no processor #, the VTune analyzer can assume a UP system and assume all samples came from cpu 0. In other cases, the VTune analyzer needs the data to actually do anything useful for analysis and presentation. For example, if there is no ip in the data samples, there is very little the VTune analyzer can usefully do with the data.	Data descriptor – An array of *data descriptor entries* that collectively define the meaning of the data in a single sample. Data descriptor entry – Defines the meaning of a unit of data within a sample. The smallest granularity for defining a unit of data in a sample is one byte. As long as certain data descriptor entries exist, the VTune analyzer can manipulate the corresponding data for analysis and presentation. Examples of data descriptor entries that can be created include: Instruction pointer, processor flags (eflags and ipsr), code segment selector, code segment descriptor, processor number, sample number, process id, thread id, privilege mode (user/kernel), execution mode (16, 32, 64 bit), timestamp counter, Event Address Registers (EAR's – instruction, data, and branch), Precise Event Buffer (PEB's) data, general purpose registers, and event id.
Event description section	An array of *event descriptor entries* that collectively describe the events that were used to	Event descriptor entry – Describes one event. Examples of event descriptive information include: event id, type of event

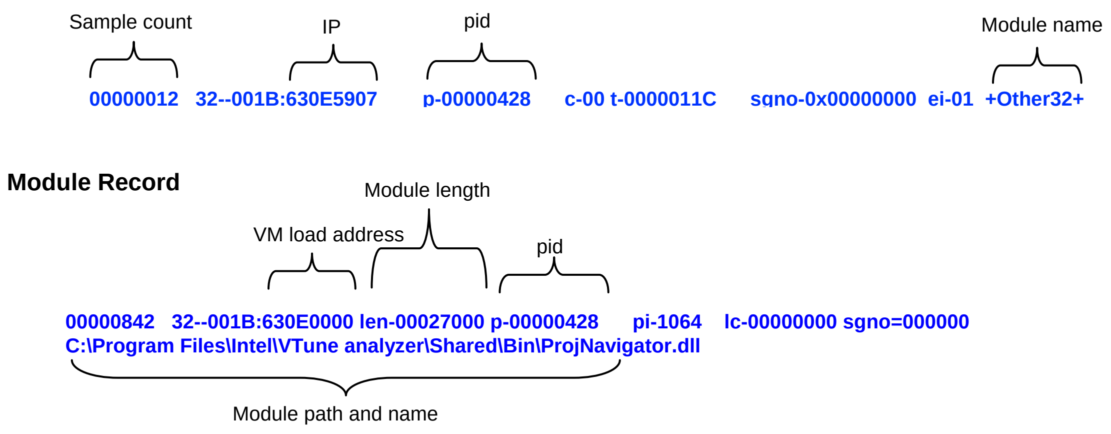
	collect the data.	(i.e. EBS vs. TBS), event name, sample after value, and event programming information. This section must exist if the data description section includes an “event id” data descriptor entry.
Data section	Contains zero or more *data entries*, where the data entries are all defined by the data descriptor in the data description section; all the data entries in a given data section have the same fixed size, which is defined by the user-specified data descriptor.	Data entry – A fixed-sized block of data whose contents are defined by a given data descriptor.

3.3 What Does “binding” Mean?

After data collection is done, binding is needed to make the data useful. Binding associates each sample record with the module, PID, and tid that it belongs to. A sample record that has not been associated with its module, PID, and tid is called a raw sample record.

For example, the module record below spans virtual memory address 630E0000 to 630E0000+27000 (i.e. 63107000) in process ID 428. The raw sample record below was collected while process ID 428 was executing and the instruction pointer of the processor was 630E5907. Since 630E0000 > 630E5907 < 63107000, the sample was mapped to the ProjNavigator.dll module in process 428.

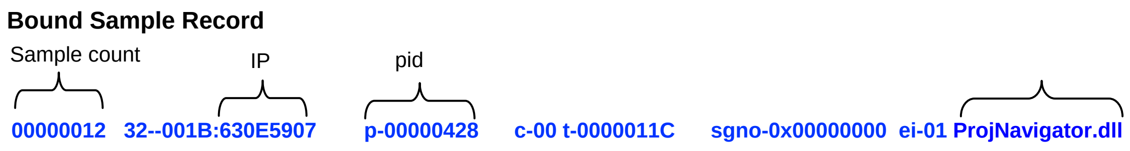
Figure 1: Sample Record



```
00000012  32--001B:630E5907    p-00000428   c-00 t-0000011C    sgno-0x00000000  ei-01  +Other32+
```

```
00000842  32--001B:630E0000 len-00027000 p-00000428    pi-1064   lc-00000000 sgno=000000
C:\Program Files\Intel\VTune analyzer\Shared\Bin\ProjNavigator.dll
```

Figure 2: Bound Sample Record



```
00000012  32--001B:630E5907   p-00000428   c-00 t-0000011C   sgno-0x00000000  ei-01 ProjNavigator.dll
```

3.4 Known Limitations

- A sample record is a fixed length record. The API does not support sample records which vary in size between samples. You can do variable sized sample records by splitting the records into a fixed portion (needed by the VTune analyzer) and a variable sized portion (which can be placed in the custom area).
- You may not be able to open an existing VTune analyzer data file and append data to one of the existing data streams.
- You may not be able to read/write/append/modify any section at any time.
- The persisted format used by the VTune analyzer may or may not be the same data format that was used by an application that collected the data.
- TBRW API supports VTune analyzer version 8.0 and higher

4 Usage Model

This section explains the several expected usage models.

4.1 Single Data Stream

Has a single set of data which is used by the VTune analyzer. For example: the current VTune analyzer.

4.2 Single Data Stream With Custom Data

Same as above but has additional user-defined data that is saved along with the information needed by the VTune analyzer. The VTune analyzer application does not interpret any data in the user-defined area. It is strictly “extra” stuff that you defined and use. The writer/reader API returns the custom data without any changes to the data itself.

For example: you have variable length data and place a “fixed length” portion in the VTune analyzer compatible data section, and any “variable length” portion in a custom section.

4.3 Multiple Data Streams With Custom Data

The basic tenant is that VTune analyzer data covers a particular span of time and the different data streams are either different pieces of data during that time or different “views” of some particular data during that time.

For example, while capturing data, the format of the data can change or there can be two, or more, different types of data being captured at the same time. Alternatively, you can aggregate the raw data and save the resulting processed data in one stream and the actual raw data in a different stream.

Example: aggregated data, bookmarks

5 Accessing a VTune™ Performance Analyzer File

This section describes the sequence of steps that you need to follow to access a VTune analyzer file using the VTune analyzer reader/writer API.

The tasks below can be performed on multiple VTune analyzer files concurrently. For example, reading from one VTune analyzer file and writing to another concurrently.

1. Optionally: Check the version of the VTune analyzer reader/writer API
   ```
   Call TBRW_get_version(...)
   ```
2. Open the VTune analyzer file.
   ```
   Call TBRW_open(...)
   ```
3. Do any of the following tasks in any order. Different tasks may be performed on a single VTune analyzer file concurrently. For example, reading two streams, and combining their data to write a third stream concurrently. However, a given section or data stream cannot be opened for read, write, or bind/unbind operations concurrently. The operations are described in the following sections in this chapter.

- Verify that the currently opened file is a valid VTune analyzer file.
  ```
  Call TBRW_verify(...)
  ```
- Write a global section.
- Read an existing global section.
- Write a data stream.
- Read an existing raw data stream.
- Read an existing bound data stream.

4. Verify that the currently opened file is a valid VTune analyzer file.
   ```
   Call TBRW_verify(...)
   ```
5. Close the VTune analyzer file.
   ```
   Call TBRW_close(...)
   ```

5.1 Writing a Global Section

Follow these steps to write a global section:

1. Call `TBRW_writing_section(section)`

   where section is a TBRW_SECTION_IDENTIFIER:
   (TBRW_HARDWARE_SECTION,

   TBRW_SOFTWARE_SECTION,

   TBRW_PROCESS_THREAD_SECTION,

   TBRW_MODULE_SECTION,

   TBRW_USER_DEFINED_GLOBAL_SECTION, or

   TBRW_VERSION_INFO_SECTION).

2. Use any of the corresponding global section access functions for writing to section, in any order. For more information, see Global Section Access.
3. Call `TBRW_done_section(section)`

5.2 Writing a Data Stream

1. Optionally: get the number of data streams currently in the file

   Call `TBRW_get_number_data_streams()`
2. Call `TBRW_writing_stream(stream_index)`

   where stream_index is a non-negative integer. The indexing starts at 0. So, for example, if there is currently one stream in the tb5 file and you want to write a second stream, you would call `TBRW_writing_stream(1)`.
3. Write to any of the stream sections for stream stream_index in any order. The sections must not exist yet – existing sections cannot be overwritten. You may write to multiple stream sections within stream_index concurrently. The stream information section and data descriptor section must be written and fully populated before the next step.
4. Call `TBRW_done_stream(stream_index)`

5.3 Writing a Stream Section

1. Call `TBRW_writing_stream_section(stream_index, section)`

   where stream_index is a non-negative integer, and section is a

   TBRW_STREAM_SECTION_IDENTIFIER:

   (TBRW_EVENT_DESCRIPTION_SECTION, TBRW_DATA_DESCRIPTION_SECTION, TBRW_DATA_SECTION,

TBRW_STREAM_INFO_SECTION, or

TBRW_USER_DEFINED_STREAM_SECTION).

2. Use any of the corresponding stream section access functions in any order, for writing to section within stream_index.
3. Call `TBRW_done_stream_section(stream_index, section)`

5.4 Writing Data With a "set_" or "add_" Function

Call the `set_` or `add_` function, passing in a pointer to the data you wish to write.

NOTE: You are responsible for any memory management that may be required – the API does not allocate or free memory for you.

5.5 Reading a Global Section

1. Call `TBRW_read_section(section)`

   where section is a TBRW_SECTION_IDENTIFIER:

   (TBRW_HARDWARE_SECTION,

   TBRW_SOFTWARE_SECTION,

   TBRW_PROCESS_THREAD_SECTION,

   TBRW_MODULE_SECTION, or

   TBRW_USER_DEFINED_GLOBAL_SECTION).
2. Use any of the corresponding global section access functions for reading from section, in any order. For more information, see Global Section Access.
3. Call `TBRW_done_section(section)`

5.6 Reading a Raw Data Stream

1. Optionally: get the number of data streams currently in the file

   Call `TBRW_get_number_data_streams()`
2. Call `TBRW_reading_stream(stream_index)`

   where stream_index is a non-negative integer.

3. Read from any of the existing stream sections for stream stream_index in any order. The sections must exist. You may read from multiple stream sections within stream_index concurrently.
4. Call `TBRW_done_stream(stream_index)`

5.7 Reading a Stream Section

1. Call `TBRW_reading_stream_section(stream_index, section)` where stream_index is a non-negative integer, and section is a TBRW_STREAM_SECTION_IDENTIFIER:
   (TBRW_EVENT_DESCRIPTION_SECTION,
   TBRW_DATA_DESCRIPTION_SECTION,
   TBRW_DATA_SECTION,
   TBRW_STREAM_INFO_SECTION, or
   TBRW_USER_DEFINED_STREAM_SECTION).
2. Use any of the corresponding stream section access functions in any order, for reading from section within stream_index. For more information see Stream Section Access.
3. Call `TBRW_done_stream_section(stream_index, section)`

5.8 Reading Data with a "get_" Function

Do one of the following:

If you already have a buffer that you think is large enough to hold the returned data:

1. Call the `get_ function`, passing in:

- the size of the buffer (IN TBRW_U32 size_of_buffer)
- a pointer to it (IN BUFFER *buf_ptr),
- Optionally, a pointer to a TBRW_U32 (OPTIONAL OUT TBRW_U32 *size_buffer_used), if you want to find out how much of the buffer was used, or NULL, if you don't.

2. Check the returned `TBRW_U32` – if it is `TBRW_BUFFER_TOO_SMALL`, then there was not enough room in your buffer to hold the returned data.

- If you passed in a non-NULL (OPTIONAL OUT TBRW_U32 *size_buffer_used), then this contains the buffer size needed.
- Or, go back to step 1, and try again.

  If you want to first check how large a buffer is required:

3. Call the get_ function, passing in:
   - 0 for (IN TBRW_U32 size_of_buffer)
   - NULL for (IN BUFFER *buf_ptr)
   - A pointer to a TBRW_U32 for (OPTIONAL OUT TBRW_U32 *size_buffer_used).
4. The TBRW_U32 pointed to by size_buffer_used contains the buffer size needed.

   Obtain a buffer at least this large, then the get_ function again, passing in:
   - the size of the buffer (IN TBRW_U32 size_of_buffer)
   - a pointer to it (IN BUFFER *buf_ptr),
   - NULL for (OPTIONAL OUT TBRW_U32 *size_buffer_used).

     `buf_ptr` now points to the data returned by the `get_ function`.

5.9 Reading Data With the enumerate_ Function

The enumerate functions rely on you to define a callback function which is called with the data requested. It is up to you to process the data. There are two prototypes for the callback function, depending on whether you need bind-related information or not.

For enumerating process info, events, data descriptors, raw data streams, raw module info, and raw thread info, the call back function must have prototype

```
TBRW_U32 (*DATA_CALLBACK)(const void *data, TBRW_U32
data_size, TBRW_U32 num_entries, void *user_ptr);
```

Where:

`data`	is a pointer to a buffer containing the data requested
`data_size`	is the size of the buffer
`num_entries`	is the number of entries contained in the data buffer
`user_ptr`	is the pointer passed into the enumerate function.

For enumerating bind-related information such as bound data streams, module binding info, and thread binding info, the call back function needs to have prototype

```
TBRW_U32 (*BIND_DATA_CALLBACK)(const void *data, TBRW_U32
sizeof_data_buffer, TBRW_U64 num_entries,
                const DATA_BIND_STRUCT *bind_table, TBRW_U32
sizeof_bind_buffer, void *user_ptr);
```

Where:

`data`	is a pointer to a buffer containing the raw data requested
`sizeof_data_buffer`	is the size of the raw data buffer
`num_entries`	is the number of entries contained in both the raw data buffer and the bind information buffer
`bind_table`	is a pointer to the bind information for the raw data
`sizeof_bind_buffer`	is the size of the bind information buffer
`user_ptr`	is the pointer passed into the enumerate function.

The bind information consists of the indexes of associated module, PID, or tid for each item in the raw buffer.

For example, when calling the `TBRW_bind_enumerate_data()` function, for each raw data sample contained in the data buffer, the `bind_table` buffer contains the index of the associated module, PID, and tid. Once you have the index, you can use the `TBRW_get_one_module/pid/tid()` functions to get the actual module, PID, or tid associated with that sample. For more information, see API Data Structure Reference.

The following are basic steps to use the enumerate functions:

1. Define a callback function as mentioned above.
2. Call the `enumerate_ function`, passing in:
   - A pointer to your callback function for (DATA_CALLBACK *my_callback_func).
   - NULL or a pointer of your choosing for `(void *user_ptr)`.
   - The start index of the entry you want to start enumerating from.
3. The `enumerate_ function` calls your callback, passing in
   - A pointer to the enumerated data(const void *data), starting from the index you specified. For example, if the start index = 0, then the enumerate_ function passes back data starting from the 1st entry. If the start index = 25, then the enumerate_ function will pass back data starting from the 26th entry.
   - The total size of the data retrieved
   - A count of how many elements are in the enumerated data, in `(TBRW_U32 num_entries)`
   - The `(void *user_ptr)` you passed to the `enumerate_ function` (in step 2b above).
4. Your callback processes the enumerated data, which is read-only.

5. Your callback should return TRUE, if you want more data, and it is available – in which case we go back to step 3. Or FALSE, if you don't need any more data, even if there are more elements available. Any return value other than TRUE (`1`) or FALSE (`0`) indicates your callback function encountered an error. In this case, the `TBRW_enumerate` function exits with the error value returned by your callback. When the `enumerate_ function` is done enumerating the data, it returns.

5.10 Binding and Unbinding a Data Stream

When viewing .tb5 files in the VTune analyzer, it automatically binds the files for you. The following API provides the programmatic interface for the same operation. You do not need to call the TBRW_dobind() explicitly.

To check if a data stream is bound:

Call `TBRW_is_bound()` on the data stream of interest. This function sets a flag that is equal zero if the data stream is not bound or non-zero if the data stream is bound.

To bind a data stream:

1. Call `TBRW_dobind()` on the data stream of interest.
2. You can call the bind information retrieval functions in any order. See the API Function Reference for complete information on the functions.
3. The following are several examples of retrieving information on bound data streams:
   - Retrieve data samples and their associated modules, PIDs, and TIDs using
     ```
     TBRW_bind_enumerate_data()
     ```
   - Retrieve modules and their associated PIDs using
     ```
     TBRW_bind_enumerate_modules()
     ```
   - Get one module record using `TBRW_get_one_module()`

To unbind a data stream:

Call `TBRW_unbbind()` on the data stream of interest.

5.11 Handling Errors

Each of the API calls returns a TBRW_U32 code. This code should be checked after each API call. When this code indicates an error has occurred:

- Optionally: Convert the error code to a string.
  `Call TBRW_error_string(...)`

- Notify the API to abort and clean up.
  `Call TBRW_abort_cleanup_and_close(...)`

6 Writing VTune™ Performance Analyzer Compatible Files

The TBRW library enables you to write your own VTune analyzer-compatible files. A VTune analyzer-compatible file is one that can be viewed by the VTune analyzer GUI. This section details the requirements for generating a VTune analyzer-compatible file.

You can use your own custom sampling collection program and view the data using the VTune analyzer. To do this, write your data using the TBRW library. This ensures that the data is in a VTune analyzer-compatible format and then view the file with the VTune analyzer viewer.

The rest of this chapter explains how to create sampling tb5 data for the VTune analyzer.

6.1 How Bind Works

During sampling collection, data samples, modules, and events are collected. During the bind process, the processes and threads are automatically generated from the data samples and modules. Processes, threads, modules, and data are written to file in an array in the tb5 file.

After binding, a mapping of the module index, pid index and tid index corresponding to each sample record in the data section. This represents the association of samples to modules indicating the source of the sample.

6.2 Required Sections

As mentioned above, during the binding process the process and thread sections are automatically deduced from the samples and the module sections. Therefore, you need not write process and thread sections.

Required Global sections:

- Hardware
- Software
- Version Info

- Modules

  Required Stream sections:

- Events
- Stream Info
- Data descriptor
- Data

  All other tb5 sections are optional, with the exception of the process and thread sections which are generated by the TBRW API and users of the TBRW API need not write these sections explicitly.

  Data for any of these sections maybe collected and saved independently (for e.g. in ring 0 driver code) and finally gather all the collected data and use TBRW API to create the tb5 file.

6.3 Hardware Section

To create hardware section, use the following data structures along with example settings.

1. Build CPUID information. This can be an array of cpuids depending on the

   architecture.

```
TBRW_CPUID_IA32 ia32_cpuid;
//all are example values
ia32_cpuid.cpuid_eax_input = 0x0;
ia32_cpuid.cpuid_eax       = 0xa;
ia32_cpuid.cpuid_ebx       = 0x756e6547;
ia32_cpuid.cpuid_ecx       = 0x6c65746e;
ia32_cpuid.cpuid_edx       = 0x49656e69;
```

2. Build CPU architecture (TBRW_CPU_ARCH)information

```
TBRW_CPU_ARCH cpu_arch;
cpu_arch.arch_size = sizeof(TBRW_CPU_ARCH);
//use TBRW_GEN_ENTRY_SUBTYPES enumeration types in
tbrw_types.h for arch_type
cpu_arch.arch_type = TBRW_GST_X86;
//number of cpuid structs available for this arch
//ia32_cpuid created above
cpu_arch.arch_num_cpuid = 1; //example value
//set the ia32_cpuid
cpu_arch.cpuid_ia32_array = (TBRW_CPUID_IA32*)&ia32_cpuid;
```

3. Build CPU (TBRW_CPU) architecture information. These can be multiple based on the number of cpus.

```
TBRW_CPU cpu_info;
cpu_info.cpu_size = sizeof(TBRW_CPU);
//cpu number
cpu_info.cpu_number = 0;
//The native architecture for this processor
//use CPU_ARCHITECTURE_IA32 or CPU_ARCHITECTURE_IA64 depending
//on the cpu architecture
cpu_info.cpu_native_arch_type = CPU_ARCHITECTURE_IA32;
//Intel processor number value (if available)
cpu_info.cpu_intel_processor_number = 8086;
//cpu speed (in Mhz)
cpu_info.cpu_speed_mhz = 3000;//example value 3GHz
//cpu front side bus speed (in Mhz)
cpu_info.cpu_fsb_mhz = 1333;//example value
//size of the L2 cache (in Kbytes)
cpu_info.cpu_cache_L2 = 4096;//example value
//size of the L3 cache (in Kbytes)
cpu_info.cpu_cache_L3 = 128;//example value
//TSC offset from cpu 0 i.e. (TSC cpu-n - TSC cpu-0)
cpu_info.cpu_tsc_offset = 0;
//package number for this cpu (if available otherwise 0)
cpu_info.cpu_package_num = 0;
//core number (if available otherwise 0)
cpu_info.cpu_core_num = 0;
//hardware thread number inside core(if available otherwise 0)
cpu_info.cpu_hardware_thread_num = 0;
//total number of h/w threads per core (if available)
cpu_info.cpu_threads_per_core = 1;
//number of cpu architectures supported
//by this cpu (from step 2 above)
cpu_info.num_cpu_arch_array = 1;
//pointer to an array of cpu architectures
cpu_info.cpu_arch_array = &cpu_arch;
```

4. Build node (TBRW_NODE) information

```
TBRW_NODE node_info;
node_info.node_size = sizeof(TBRW_NODE);
node_info.node_type_from_shell = 0;//set it to 0
node_info.node_id = 1;//node id
//total number of cpus in the system – cpu_info structure
above
node_info.node_num_available = 1;
//number of cpus used based on cpu-mask specified
node_info.node_num_used = 1;
//physical memory on the node in MBytes
node_info.node_physical_memory = 1024;
//cpu array – address of cpu_info structure created above
node_info.cpu_array = &cpu_info;
```

5. Build system (TBRW_SYSTEM) information.

```
TBRW_SYSTEM sys_info;
sys_info.system_size = sizeof(TBRW_SYSTEM);
//number of node arrays
sys_info.system_num_nodes = 1;
//address of node array
sys_info.node_array = &node;
```

6.4 Software Section

To create software section, use the following data structures along with example settings.

1. Build host information like IP address and name and call TBRW_set_host()

```
TBRW_HOST host_info;
host_info.host_size = sizeof(TBRW_HOST);
//set the ip address of the host system
host_info.host_ip_address.soi_ptr = (TBRW_CHAR*)
_wcsdup(L"10.10.10.10");
//set the host name of the host system
host_info.host_name.soi_ptr = (TBRW_CHAR*)
_wcsdup(L"testhost.intel.com");
ret = TBRW_set_host(tbrw_ptr, &host_info);
```

2. Build operating system information and call TBRW_set_os()

```
TBRW_OS os_info;
os_info.os_size = sizeof(TBRW_OS);
//set to one of TBRW_OSFAMILY_ types
os_info.os_platform = TBRW_OSFAMILY_WIN32;
//OS major number
os_info.os_major = 0x5;//example value
```

```
//OS minor number
os_info.os_minor = 0x1;//example value
//OS build number
os_info.os_build = 0xa28;//example value
//OS extra information like service packs, etc.
os_info.os_extra = 0;
//OS name
os_info.os_name.soi_ptr = (TBRW_CHAR*) _wcsdup(L"Windows XP");
//OS name extra information
os_info.os_name_extra.soi_ptr = (TBRW_CHAR*) _wcsdup(L"32-
bit");
ret = TBRW_set_os(tbrw_ptr, &os_info);
```

3. Build application information and call TBRW_set_application().

```
TBRW_APPLICATION app_info;
//system application specific data like page size, etc.
app_info.app_size = sizeof(TBRW_APPLICATION);
app_info.app_page_size = 1024;//example value
app_info.app_alloc_granularity = 2048;//example value
app_info.app_min_app_addr = 0xffff;//example value
app_info.app_max_app_addr = 0xffffffff;//example value
ret = TBRW_set_application(tbrw_ptr, &application);
```

6.5 Version Info Section

To create version info section, use the following data structures along with example settings.

1. Build version information and call TBRW_set_version_info()

```
TBRW_VERSION_INFO version_info;
version_info.version_info_size = sizeof(TBRW_VERSION_INFO);
//set the sampling_driver version to 0xffffffff
version_info.sampling_driver_version = 0xffffffff;
//set bind version to 0
version_info.bind_version = 0;
//set sampling file version
version_info.sample_file_version = 0x10;//example value
ret = TBRW_set_version_info(tbrw_ptr, &version_info);
```

6.6 Module Section

To create module section, capture the information of all the system and application modules loaded by the running processes on the system.

1. Build module information for each captured module and call TBRW_add_module().

   Use the following values for some of the fields.

   How to set module_flags:
   `module_info.module_flags = TBRW_MODULE_TSC_USED | <`*`other combination of flags depending on the module as described below`*`>`

   TBRW_MODULE_LOADED - for load notification of module,
   TBRW_MODULE_UNLOADED - for unload notification of module
   TBRW_MODULE_TSC_USED - time stamps are used for module load/unload events
   TBRW_MODULE_IS_EXE - if the module is an executable
   TBRW_MODULE_GLOBAL_LOAD. - if the module is global
   TBRW_MODULE_1ST_MODULE_REC_IN_PROCESS - if the module is the first notification in a process

   How to set module code selector:
   `module_info.module_code_selector = <`*`code selector value`*`>`
   The <*`code selector value`*> varies depending on the OS, architecture and type of the module (32/64 bit and kernel/user mode)
   For example:
   - On 32-bit, NT ring 0 module ->0x8
     ring 3 module ->0x1B
   - On 64-bit, NT ring 0 64-bit module -> 0x10
     ring 3 64-bit module -> 0x33
     ring 3 32-bit module -> 0x23
   - On Linux, values like __KERNEL_CS and __USER_CS are used.

   So, it depends on your environment.

NOTE: This code selector value should match the code selector value used in sample record (TBRW_SampleRecordPC.cs field).

How to set module segment type:
`module_info.module_segment_type = <`*`mode type`*`>`, where <*`mode type`*> is one of the following:
TBRW_MODE_32BIT - for 32-bit module or 32-bit module on 64-bit OS's
TBRW_MODE_64BIT - for 64-bit module

How to set module load address (in memory):
`module_info.module_load_address = starting address` where module is loaded in memory. This field together with `module_length` indicates the memory range where module is loaded.
module_info.module_length = length from the starting address

```
TBRW_MODULE module_info;
module_info.module_size = sizeof(TBRW_MODULE);

module_info.module_load_address = 0x400000;//example value
module_info.module_length = 0x2000;//example value
//PID of the process in this module is loaded
module_info.module_associated_pid = 0x1000;//example value
//code selector specific to the platform/architecture as
described above
module_info.module_code_selector = 0x8;//example value
//time stamp when module load occurred, set to 0 if
//module is already loaded before the collection started
module_info.module_event_when = (TBRW_U64)0x0000;//example
value
//time stamp when module is unloaded. Set to
//-1 if module continues to be loaded even
//after the collection stopped
module_info.module_unload_tsc = (TBRW_U64)-1;//example value
//set to TBRW_MODE_32BIT or TBRW_MODE_64_BIT
module_info.module_segment_type = TBRW_MODE_32BIT;//32-bit
//set to zero
module_info.module_segment_number = 0;
module_info.module_flags = TBRW_MODULE_TSC_USED |
TBRW_MODULE_LOADED | TBRW_MODULE_IS_EXE |
TBRW_MODULE_1ST_MODULE_REC_IN_PROCESS;
module_info.module_path.soi_ptr =
(TBRW_CHAR*)_wcsdup(L"C:\\sample\\sample.exe");//example value
module_info.module_name.soi_ptr =
(TBRW_CHAR*)_wcsdup(L"sample.exe");//example value
module_info.module_segment_name.soi_ptr =
(TBRW_CHAR*)_wcsdup(L"");
//set all the following fields to 0
module_info.reserved_for_legacy = 0;
module_info.reserved_for_legacy2 = 0;
module_info.reserved_for_legacy_flags = 0;
ret = TBRW_add_module(tbrw_ptr, &module_info);
```

6.7 Events Section

To create events section, use the following data structures along with example settings.

1. Build host information like IP address and name and call TBRW_set_host()

```
TBRW_EVENT_DETAIL event_det;
event_det.detail_size         = sizeof(TBRW_EVENT_DETAIL);
//detail access is read or write access –
//read = 0 and write = 1
event_det.detail_access       = 1;
//set to size of access in bits – 8/16/32/64
event_det.detail_access_size  = 32;
//use values from TBRW_ACCESS_METHOD
//enumeration in tbrw_types.h for detail_method
event_det.detail_method       = TBRW_MSR;
//value programmed
//address of read/write
event_det.detail_address      = 0x21;//example value
event_det.detail_value        = 0xffe17b80;//example value
//set legacy_command to 0
event_det.legacy_command      = 0;
```

2. Build event information and call TBRW_add_event() for each event that the sampling data is collected for.

```
TBRW_EVENT event_info;
event_info.event_size        = sizeof(TBRW_EVENT);
//event id – this map to the event id values used in
TBRW_SampleRecordPC.eventIndex field
event_info.event_id          = 0;//event sequence index
//number of event details (TBRW_EVENT_DETAIL structure)
event_info.event_num_details = 1;//e.g. only one event
profiled
//set to TBRW_EVENT_EBS
event_info.event_flags = TBRW_EVENT_EBS;
//set to the sample after value used for the event
event_info.event_sav = 2000000;//example sav value
//set to the event name (human readable name)
event_info.event_name.soi_ptr = (TBRW_CHAR*)
_wcsdup(L"SampleEvent");
//set to the event archicture name (human readable name)
event_info.event_arch_name.soi_ptr = (TBRW_CHAR*)
_wcsdup(L”Core 2");
//set to address of event_det array
event_info.event_detail_array = &event_det;
//2nd parameter is stream = 0
```

```
ret = TBRW_add_event(tbrw_ptr, 0, & event_info);
```

6.8 Stream Info Section

To create stream info section, use the following data structures along with example settings.

1. Build stream information and call TBRW_set_stream_info()

```
TBRW_STREAM_INFO stream_info;
stream_info.stream_info_size  = sizeof(TBRW_STREAM_INFO);
//sampling duration in milli-seconds
stream_info.sampling_duration = 20000;//example value
// stream type, in this case sampling stream
stream_info.stream_type       = TBRW_SAMPLING_STREAM;
//set to the command line used for collecting sampling data
stream_info.command_line.soi_ptr =
(TBRW_CHAR*)_wcsdup(L"unknown");
stream_info.comment.soi_ptr =
(TBRW_CHAR*)_wcsdup(L"sample_comment");
stream_info.cpu_mask.soi_ptr = (TBRW_CHAR*)_wcsdup(L"none");
//sampling interval in milli-seconds
stream_info.sampling_interval = 1000;//example value
//sampling data collection start time – example value below
stream_info.sampling_start_time.dwLowDateTime  = 1526932352;
stream_info.sampling_start_time.dwHighDateTime = 29849313;
//sampling data collection stop time – example value below
stream_info.sampling_end_time.dwLowDateTime  = 1566932352;
stream_info.sampling_end_time.dwHighDateTime = 29849313;
ret = TBRW_set_stream_info(tbrw_ptr, 0, &stream_info);
```

6.9 Sampling Data Descriptors

To create sampling data section that you can view with the VTune analyzer, use the following data descriptor types for creating TBRW_SAMPREC_DESC_ENTRY entries.

1. Add sample record descriptor

```
TBRW_SAMPREC_DESC_ENTRY desc_entry1;
desc_entry1.desc_size      = sizeof(TBRW_SAMPREC_DESC_ENTRY);
//Size of the data being described
desc_entry1.desc_data_size = sizeof(TBRW_SampleRecordPC);
desc_entry1.desc_offset    = 0;
```

```
//set desc_type to ST_LEGACY_SAMPLE_RECORD
desc_entry1.desc_type      = ST_LEGACY_SAMPLE_RECORD;
//set desc_sub_type to SST_NONE
desc_entry1.desc_sub_type  = SST_NONE;
//set desc_sample_flag to 0
desc_entry1.desc_sample_flag = 0;
desc_entry1.desc_name.soi_ptr = (TBRW_CHAR*)
_wcsdup(L"SampleRecord");//human readable name
ret = TBRW_add_data_descriptor_entry(tbrw_ptr, 0,
&desc_entry1);
```

2. Add TSC descriptor

```
TBRW_SAMPREC_DESC_ENTRY desc_entry2;
desc_entry2.desc_size = sizeof(TBRW_SAMPREC_DESC_ENTRY);
desc_entry2.desc_data_size = sizeof(TBRW_SampleRecordTSC);
desc_entry2.desc_offset   = 0;
//set desc_type to ST_TIMESTAMP
desc_entry2.desc_type     = ST_TIMESTAMP;
//set desc_sub_type to SST_TSC
desc_entry2.desc_sub_type = SST_TSC;
//set desc_sample_flag to 0
desc_entry2.desc_sample_flag  = 0;
desc_entry2.desc_name.soi_ptr = (TBRW_CHAR*)
_wcsdup(L"TimeStamp);
ret = TBRW_add_data_descriptor_entry(tbrw_ptr, 0,
&desc_entry2);
```

6.10 Sample Record Data Structure

The sampling data needs to be in a specific format for the VTune analyzer viewer to display the TB5 files. The sampling data must be written to data stream 0. Each sample record needs to fit into the TBRW `SampleRecordPC` structure, defined in `tbrw_types.h` header file as:

```
typedef struct TBRW_SampleRecordPC_s {   // Program Counter section
   union {
      struct {
         TBRW_U64 iip; // IA-64 architecture interrupt instruction pointer
         TBRW_U64 ipsr; // IA-64 architecture interrupt processor status
register (eflags)
             };
      struct {
         TBRW_U32  eip;          // IA-32 architecture instruction pointer
         TBRW_U32  eflags;       // IA-32 architecture eflags
         TBRW_CodeDescriptor csd;// IA-32 architecture code seg descriptor
(8 bytes)
             };
          };
    TBRW_U16    cs;            // IA-32 architecture code segment (0 for IA-
64 architecture)
    union {
      TBRW_U16 cpuAndOS;        // cpu and OS info as one word
      struct {                  // cpu and OS info broken out
        TBRW_U16 cpuNum : 12;   // cpu number (0 - 4096)
        TBRW_U16 notVmid0 : 1;  // not being used, set to zero
        TBRW_U16 codeMode : 2;  // not being used, set to zero
        TBRW_U16          : 1;  // reserved
             };
          };
    TBRW_U32   tid;         // thread ID
    TBRW_U32   pidRecIndex; // process ID
    union {
      TBRW_U32 bitFields2;
      struct {
        TBRW_U32   mrIndex : 20;   // module record index (index into start
of module rec section)
        TBRW_U32   eventIndex : 8; // index into the Events section of the
event that triggered this sample
        TBRW_U32   tidIsRaw : 1;   // not being used, set to zero
        TBRW_U32   IA64PC : 1;     // IA-64 architecture PC sample
(TRUE=this is a IA-64 architecture PC sample record)
        TBRW_U32   pidRecIndexRaw : 1; // pidRecIndex is raw OS pid
        TBRW_U32   mrIndexNone : 1;    // no mrIndex (unknown module)
             };
          };
} TBRW_SampleRecordPC, *TBRW_PSampleRecordPC;
```

You need to write raw data and let the TBRW API do the binding. As mentioned in the previous section, when a tb5 file is viewed using VTune analyzer, the file automatically goes through the bind operation. There is no need to explicitly call the TBRW bind API.

To add sampling data section:

1. Build a sample record and add using TBRW_add_data().

NOTE: You can alternatively create a temp file during sampling data collection as per the expected format (TBRW_SampleRecordPC and TBRW_SampleRecordTSC) and call TBRW_add_data_from_file() function.

Use the following values for some of the fields. Note that the sample.iip and sample.isr combination is for IPF architecture. The sample.eip, sample.eflags and sample.csd combination is for other architectures (IA32, EM64T).

To set the code selector:

Set the code selector (cs) value similar to module:
sample.cs = <code selector value>
The <code selector value> varies depending on the OS, architecture and type of the module (32/64 bit and kernel/user mode). For example:

- On 32-bit, NT ring 0 module ->0x8
  ring 3 module ->0x1B
- On 64-bit, NT ring 0 64-bit module -> 0x10
  ring 3 64-bit module -> 0x33
  ring 3 32-bit module -> 0x23
- On Linux, values like __KERNEL_CS and __USER_CS are used.

So, it depends on your environment.

NOTE: This code selector value should match the code selector value used in sample record (TBRW_SampleRecordPC.cs field).

To set codeMode:

`sample.codeMode  = <mode type>`, where *<mode type>* is one of the following:

TBRW_MODE_32BIT - for 32-bit module or 32-bit module on 64-bit OS's

TBRW_MODE_64BIT - for 64-bit module

To set eventIndex:

set eventIndex = event id of the event (in events section) for which this sample is collected, this is the sequential event number (0, 1, 2...)

To set the IA32/EM64T details:

a. Set sample.eip to IA-32 Extended instruction pointer

b. Set sample.eflags to IA-32 architecture eflags from interrupt frame

c. Set sample.csd to IA-32 architecture code segment from interrupt frame

To set the IA64 details:

a. Set sample.iip to IA64 interrupt Instruction Pointer

b. Set sample.ipsr to IA64 interrupt processor status register

```
TBRW_SampleRecordPC sample;
//use TBRW_MODE_ types
sample.codeMode       = TBRW_MODE_32BIT;//32-bit
```

```
//set eip to the extended instruction pointer
//note that iip and ipsr need to be set for IPF
//and eip/eflags and csd need to be set for others
sample.eip            = 0x617000;//example value
//set eflags to IA32 architecture eflags
sample.eflags         = 0x246;//example value
//set csd to IA32 architecture code segment
sample.csd.lowWord    = 0xffff;//example value
sample.csd.highWord   = 0xcf9b00;//example value
//set cs to the code selector
//set cs to 0 for IPF
sample.cs             = 0x1B;
//set it to zero, this will be filled-in later
sample.mrIndex        = 0;
//set it to same as the event_id field in TBRW_EVENT
sample.eventIndex     = 0;//In this case event 0
//set tidIsRaw to 1
sample.tidIsRaw       = 1;
//set IA64PC to 1 for IPF architecture and 0 for others
sample.IA64PC         = 0;
//set pidRecIndexRaw to 1
sample.pidRecIndexRaw = 1;
//set pidRecIndex to the PID for which the sample collected
sample.pidRecIndex    = 0x1000;//example value
//set cpuNum to cpu number for which the
//sampling interrupt is genearted
sample.cpuNum         = 0;
//set tid to the thread id value
sample.tid            = 0;
//set mrIndexNone to 0
sample.mrIndexNone    = 0;
ret = TBRW_add_data(tbrw_ptr, 0, sizeof(TBRW_SampleRecordPC),
(void*)&sample);
```

2. Build TSC data and call TBRW_add_data to call

```
TBRW_SampleRecordTSC tsc;
Tsc.tsc = 0x00001972178D8F5E;
ret = TBRW_add_data(tbrw_ptr, 0, sizeof(TBRW_SampleRecordTSC),
(void*)&tsc);
```

6.11 64-bit Samples vs. 32-bit Samples

If you are collecting 64-bit samples, you need to fill out the `iip` and `ipsr` fields, and set the `IA64PC field = 1`.

If you are collecting 32-bit samples, you need to fill out the `eip`, `csd` and `eflags` field, set the `IA64PC field = 0`.

7 FAQ

7.1 The os_platform field in the TBRW_OS structure is an integer and described as an enumerated type. Do we use a generic "other" indicator ?

The operating system platform is defined in tbrw_types.h header file as:

```
#define TBRW_OSFAMILY_WIN32     0x00000001
#define TBRW_OSFAMILY_WIN64     0x00000002
#define TBRW_OSFAMILY_WINCE     0x00000003
#define TBRW_OSFAMILY_XOS       0x00000004
#define TBRW_OSFAMILY_LINUX32   0x00000005
#define TBRW_OSFAMILY_LINUX64   0x00000006
```

You need to define your operating system platform type as any number between the range 0x100 and 0x1FF. All other bytes are reserved for internal use.

7.2 32 bit PIDs will not be sufficient for 64 bit OS, should be TBRW_U64.

If you want to write unbound data to the TB5 file, and apply our binding algorithm to generate the PIDs and TIDs, then you only need to write the data samples, modules, and events section (as described in "Writing VTune Performance Analyzer compatible files" section). The PIDs and TIDs are generated during the bind process. In this case, the TBRW API adjusts the data structures to handle 64 bit PIDs but as you see above in the TBRW_SampleRecordPC definition, the bind algorithm currently handles only 32 bit PIDs and TIDs. Therefore, there would need to be some translation from 64 bit PIDs to 32 bit PIDs when writing the sampling data. You need to do this translation yourself.

7.3 Same applies for TID

The above also applies for TID.

7.4 In TBRW_VERSION_INFO structure, what value should we use for the sampling_driver field? Do we use "other"?

Set it to 0xFFFFFFFF, which indicates "unknown" type.

7.5 Event mapping will require a bit more detail. I don't see much I recognize

Depending on what the goal is, not all fields in the events section need to be filled in. If the objective is to be able to write a TB5 file, and then import the file into the VTune analyzer for viewing, then only the `event_size`, `event_id`, `event_flags`, `event_sav`, and `event_name` need to be filled in. The `event_num_details`, `event_arch_name`, and `event_detail_array` can be set to zero. The `event_detail_array` is used to record in the TB5 file how the registers were programmed in order to collect on those events. Its presence has no effect on viewing. Please note that the event_id values in this section should be used as the eventIndex field in TBRW_SampleRecordPC structure.

7.6 Why does my call to TBRW_convert_uniqueid_to_string() sometimes return an "invalid string" error? Is this expected?

This may happen when the function is trying to convert a string with unique id = 0, in other words a NULL string. This is expected behavior and indicates that the string is invalid. Check to see if the string's unique id is zero.

7.7 Where can I find a list of error return codes?

All error codes are available in the public header file `tbrw.h`

8 API Data Structure Reference

The types used in defining the API and the data structures below specify the types in an OS and compiler independent manner. The types are:

- U? indicates unsigned. The ? is a number indicating the number of bits. For example, TBRW_U32 indicates a 32-bit unsigned value
- S? indicates signed. For example S64 indicates a signed 64-bit value.

A header file, provided as part of this release, sets up the proper typedefs to make this true on a variety of environments.

The VTune analyzer reader/writer API expects all strings to be UNICODE. When writing strings to disk, the UNICODE strings are converted to multibyte for backwards compatibility with VTune analyzer legacy readers. Where applicable, the API's use strings of type TBRW_CHAR, which is #defined to be `wchar_t`. This enables changing the string type at a later date.

The TBRW library accepts pointers to string for write operations. For read operations, the value retuned is a unique string identifier which can be converted back to the original string on demand. This enables the library to return information (strings, records, whatever) regardless of the on-disk size.

In general, the data structures are defined so that compiler padding is either minimized or explicitly part of the structure. This facilitates porting code between different operating systems, architectures, and compilers.

NOTE: Do not assume that the TBRW structures in this document (which are defined in tbrw_types.h file) are similar to the internal structures.

8.1 Basic Types

The following are the basic TBRW data types.

```
typedef     wchar_t             TBRW_CHAR;
typedef     unsigned char       TBRW_U8;
typedef     unsigned short      TBRW_U16;
typedef     unsigned int        TBRW_U32;
typedef     signed int          TBRW_S32;
#if defined(TBRW_OS_LINUX) || defined(TBRW_OS_APPLE)
typedef     signed long long    TBRW_S64;
typedef     unsigned long long  TBRW_U64;
#elif defined(TBRW_OS_WINDOWS)
```

```
typedef      signed __int64       TBRW_S64;
typedef      unsigned __int64     TBRW_U64;
#endif

typedef void *TBRW_PTR;    // as far as the user is concerned

typedef struct __string_or_id {  // on writes, it is a string pointer
                                          // on reads, it is a unique id.
    union {
        TBRW_U64       soi_uniqueid;        // on reads, it is a unique id.
        TBRW_CHAR *soi_ptr;        // for debug/implementation can use
                                    // bit 63 to indicate whether it is an
unique id or not
    };
} TBRW_STRING_OR_ID;
```

8.2 Section Identifiers

```
typedef enum {
    TBRW_HARDWARE_SECTION,
    TBRW_SOFTWARE_SECTION,
    TBRW_PROCESS_THREAD_SECTION,
    TBRW_MODULE_SECTION,
    TBRW_USER_DEFINED_GLOBAL_SECTION,
    TBRW_VERSION_INFO_SECTION
} TBRW_SECTION_IDENTIFIER;

typedef enum {
    TBRW_EVENT_DESCRIPTION_SECTION,
    TBRW_DATA_DESCRIPTION_SECTION,
    TBRW_DATA_SECTION,
    TBRW_USER_DEFINED_STREAM_SECTION,
    TBRW_STREAM_INFO_SECTION
} TBRW_STREAM_SECTION_IDENTIFIER;
```

8.3 Hardware Structures

```
typedef struct __TBRW_system {
    TBRW_U32 system_size;  // set to sizeof(TBRW_SYSTEM). Used for versioning
    TBRW_U32 system_num_nodes;// number of nodes in a system
    TBRW_NODE *node_array;// pointer to an array of nodes for this system
} TBRW_SYSTEM;

typedef struct __TBRW_node {
    TBRW_U32 node_size;    // set to sizeof(TBRW_NODE). Used for versioning
    TBRW_U32 node_type_from_shell;      //the shell platform
    TBRW_U32 node_id;                   // The node number/id (if known)
    TBRW_U32 node_num_available;        // total number cpus on this node
    TBRW_U32 node_num_used;             // number used based on cpu mask at
time of run
    TBRW_U64 node_physical_memory;      // amount of physical memory on this
node
    TBRW_CPU *cpu_array;  // pointer to an array of cpu’s for this node
} TBRW_NODE;


typedef struct __TBRW_cpu {
    TBRW_U32 cpu_size;     // set to sizeof(TBRW_CPU). Used for versioning
    TBRW_U32 cpu_number;              // The cpu number
    TBRW_U32 cpu_native_arch_type;    // The native architecture for this
processor
    TBRW_U32 cpu_intel_processor_number; // The intel processor number (if
available)
    TBRW_U32 cpu_speed_mhz;             // cpu speed (in Mhz)
    TBRW_U32 cpu_fsb_mhz;               // cpu front side bus speed (in Mhz)
    TBRW_U32 cpu_cache_L2;
// Size of the L2 cache (in Kbytes)
    TBRW_U32 cpu_cache_L3;
// Size of the L3 cache (in Kbytes)
    TBRW_S64 cpu_tsc_offset;
// TSC offset from CPU 0 ie. (TSC CPU N – TSC CPU 0)
    TBRW_U16 cpu_pacakge_num;
// package number for this cpu (if known)
    TBRW_U16 cpu_core_num;
// core number (if known)
    TBRW_U16 cpu_hardware_thread_num;
// hardware thread number inside core (if known)
    TBRW_U16 cpu_threads_per_core;
// total number of h/w threads per core (if known)
    TBRW_U32 num_cpu_arch_array;
//number of cpu architectures supported by this cpu


    TBRW_CPU_ARCH *cpu_arch_array;
// pointer to an array of cpu
// architectures supported by this cpu (for
// example, IA-64 architecture processors that support execution
// of IA-32 architecture binaries can have two elements in the
// cpu_arch_array table). The native architectural
// type must be represented in this array.
} TBRW_CPU;

typedef struct __TBRW_cpu_arch {
    TBRW_U32 arch_size;
// set to sizeof(TBRW_CPU_ARCH). Used for versioning
    TBRW_U16 arch_type;
// enum of architecture (IA-32, IA-64, Intel® 64).
//the enumeration is defined in the header file
//samp_info.h and is called GEN_ENTRY_SUBTYPES
    TBRW_U16 arch_num_cpuid;  // number of cpuid structs available for this
arch
    union {
        TBRW_CPUID_IA32 *cpuid_ia32_array;
        TBRW_CPUID_IA64 *cpuid_ia64_array;
    };
} TBRW_CPU_ARCH;

typedef struct __TBRW_cpuid_ia32 {
    TBRW_U32 cpuid_eax_input;
    TBRW_U32 cpuid_eax;
    TBRW_U32 cpuid_ebx;
    TBRW_U32 cpuid_ecx;
    TBRW_U32 cpuid_edx;
    TBRW_U32 reserved
} TBRW_CPUD_IA32;

typedef struct __TBRW_cpuid_ia64 {
    TBRW_U64 cpuid_select;
    TBRW_U64 cpuid_val;
    TBRW_U64 reserved;
} TBRW_CPUID_IA64;
```

8.4 Software Structures

```
typedef struct __TBRW_host {
    TBRW_U32 host_size;                        // set to sizeof(TBRW_HOST).
Used for versioning
    TBRW_STRING_OR_ID host_ip_address;   // IP address of the host
    TBRW_STRING_OR_ID host_name;        // human readable host name
    TBRW_U32 reserved;
} TBRW_HOST;

typedef struct __TBRW_os {
    TBRW_U32 os_size;            // set to sizeof(TBRW_OS). Used for 
versioning
    TBRW_U32 os_platform;        // OS indicator (linux, windows, etc)
    TBRW_U32 os_major;           // OS major version
    TBRW_U32 os_minor;           // OS minor version
    TBRW_U32 os_build;           // OS build number
    TBRW_U32 os_extra;           // OS release info, service packs, errata 
numbers, etc.
    TBRW_STRING_OR_ID os_name;   // human readable OS name
    TBRW_STRING_OR_ID os_name_extra;// human readable OS arbitrary extra 
information (like
// service packs, errata, the result of `uname -r`, etc)
} TBRW_OS;

typedef struct __TBRW_application {
    TBRW_U32 app_size;           // set to sizeof(TBRW_APPLICATION). Used 
for versioning
    TBRW_U32 app_reserved;       // reserved, should be set to zero
    TBRW_U32 app_page_size;      // page size (as seen by application)
    TBRW_U32 app_alloc_granularity;// granularity of vm (i.e. mmap() 
size/alignement)
    TBRW_U64 app_min_app_addr;   // lowest memory address accessible by an 
application
    TBRW_U64 app_max_app_addr;   // highest memory address accessible by an 
application
} TBRW_APPLICATION;
```

8.5 Process/Thread Structures

```
typedef struct __TBRW_pid {
    TBRW_U32 pid_size;     // set to sizeof(TBRW_PID). Used for versioning
    TBRW_U32 pid_reserved; // reserved, should be zero
    TBRW_U32 pid_id;       // process id (as provided by the OS). Needs
                           // to be comparable against the PID field in the 
sample record

    TBRW_U32 pid_flags;    // Creation or termination event, event_when is
                          // tsc or sample number, etc.
    TBRW_U64 pid_event_when;// An indication of when the pid event occured
(i.e.
                           // could be a tsc value, could be a sample number,
etc.)
    TBRW_U32 reserved1;       // can be used for cpu # later on when we move
to tsc's
    TBRW_U32 reserved2;
    TBRW_STRING_OR_ID pid_path;          // path to the process executable
(if create event)
    TBRW_STRING_OR_ID pid_name;          // name of the process (if create
event)
} TBRW_PID;


typedef struct __TBRW_tid {
    TBRW_U32 tid_size;                   // set to sizeof(TBRW_TID). Used
for versioning
    TBRW_U32 tid_associated_pid; // process that this thread is a part of.
Needs
                            // to be comparable against the PID field in the
                            // sample record and the TBRW_PID pid_id field
    TBRW_U32 tid_id;       // thread id (as provided by the OS). Needs
                           // to be comparable against the TID field in the
                           // sample record
    TBRW_U32 tid_flags;    // Creation or termination event, event_when is
                           // tsc or sample count, etc.
    TBRW_U64 tid_event_when;// An indication of when the tid event occured
                            // (i.e could be a tsc value, could be a sample
                            // number, etc)
    TBRW_U32 reserved1;    // can be used for cpu # later on when we move to
tsc's
    TBRW_U32 reserved2;
    TBRW_STRING_OR_ID tid_name;  // name of the thread (if create event)

} TBRW_TID;
```

8.6 Module Structure

```
typedef struct __TBRW_module {
    TBRW_U32 module_size;        // set to sizeof(TBRW_MODULE). Used for
versioning
    TBRW_U32 module_reserved;    // reserved, should be zero
    TBRW_U32 module_associated_pid;// process which loaded/unloaded this
module
    TBRW_U32 module_flags;        // addition information about this module
(load
                                  // vs. Unload, global,
                                  // exe, segment type, event_when is tsc
                                  // vs. Sample count, etc).


    TBRW_U64 module_event_when;  // An indication of when the event
occurred. Could be
                                 // a timestamp or correspond to a
particular sample
                                 // (sample number)
    TBRW_U32 module_segment_number;// for java
    TBRW_U32 module_segment_type : 2;// see the MODE_ types defined in
SampFile.h
    TBRW_U32                     :30;
    TBRW_U32 module_code_selector;// for IA-32 architecture


    TBRW_U64 module_length;      // size of the module (if load event)
    TBRW_U64 module_load_address;// address where module was loaded (if load
event)
    TBRW_U32 reserved_for_legacy;// holds module unload sample count, please
don't use
    TBRW_U32 reserved_for_legacy2;// for now holds the pid index in the case
// of a  bound tb5 file
                                 // WARNING: temporary only!
    TBRW_U32 reserved_for_legacy_flags; // same as legacy ModuleRecord.flags
field
    TBRW_U32 reserved1;          // can be used for cpu # later on
    TBRW_U64 module_unload_tsc;  // Saves the load tsc for tsc based data
collection,
                                 // module_event_when is used for old sample
                                 // sample count based tb5 data
    TBRW_STRING_OR_ID module_path;// path to the module (if load event)
    TBRW_STRING_OR_ID module_name;// name of the module (if load event)
    TBRW_STRING_OR_ID module_segment_name;// name of the segment (if load
event and segments in use)


} TBRW_MODULE;
```

8.7 Version Information Structure

```
typedef struct __TBRW_version_info
{
    TBRW_U32 version_info_size;        //set to sizeof(TBRW_VERSION_INFO)
    TBRW_U32 sampling_driver_version;  //version of the sampling driver
    TBRW_U32 bind_version;             //version of the bind library used to
analyze the tb5 file
    TBRW_U32 sample_file_version;      //version of the sample file format

} TBRW_VERSION_INFO;
```

8.8 Stream Information Structure

```
typedef struct __TBRW_stream_info
{
    TBRW_U32 stream_info_size;    //set to sizeof(TBRW_STREAM_INFO)
    TBRW_U32 sampling_duration;   //duration of the sampling session (in
milliseconds ??)
    TBRW_U32 stream_type;         //refer to TBRW_STREAM_TYPE
    TBRW_U32 sampling_interval;   //in microseconds
    VTUNE ANALYZER_FILETIME sampling_start_time;
                                  //start time of the sampling session
    VTUNE ANALYZER_FILETIME sampling_end_time;
                                 //end time of the sampling session
    TBRW_STRING_OR_ID command_line;
                           //the command line used to generate this stream
    TBRW_STRING_OR_ID cpu_mask;
//even though the cpu mask info is also found in the command line,
//we include it here too because TBRW does not
//know how to parse the command line
//(could be sep command, vtl command, dcpi
 //command, etc).
    TBRW_STRING_OR_ID comment;  //the comment describing this stream

} TBRW_STREAM_INFO;

//Note that one usage model for filling in the stream info structure is
//to set stream_type = TBRW_CUSTOM_STREAM and set the comment to a string of
your choice.
//Then, when reading the tb5 file to find your custom stream, you can
iterate through
```

```
//all streams and parse the comment field to find the right stream
```

8.9 Stream Types

```
typedef enum {
    TBRW_SAMPLING_STREAM = 1,
    TBRW_AGGREGATED_STREAM,
    TBRW_BOOKMARK_STREAM,
    TBRW_BIND_DATA_INFORMATION_STREAM,
    TBRW_BIND_MODULE_INFORMATION_STREAM,
    TBRW_BIND_PID_INFORMATION_STREAM,
    TBRW_BIND_TID_INFORMATION_STREAM,
    TBRW_CUSTOM_STREAM
} TBRW_STREAM_TYPE;
```

8.10 Event Descriptor

An event descriptor is an array of TBRW_EVENT that describes the events that were used to collect the data in a corresponding data stream.

```
//Note: an array of type TBRW_EVENT_DETAIL immediately follows each
TBRW_EVENT
//The array has event_num_details number of items in it
typedef struct __TBRW_event {
    TBRW_U32 event_size;   //set to sizeof(TBRW_EVENT). Used for versioning
    TBRW_U32 event_id;     //event id, a sequential index (0, 1, 2 …).
                          //Must be comparable to the event id
                         //field  in the sample record.
    TBRW_U32 event_num_details;// number of event details supplied
    TBRW_U32 event_flags;  //event info (ie. Ebs vs. Tbs, units for the SAV,
etc)
    TBRW_U64 event_sav;    // sample after value used for this event
    TBRW_STRING_OR_ID event_name;// human readable name of the event
    TBRW_STRING_OR_ID event_arch_name;  //human readable name of cpu type,
i.e. "Pentium M"
    TBRW_EVENT_DETAIL *event_detail_array; //details about programming this
event

} TBRW_EVENT;

typedef struct __TBRW_event_detail {
    TBRW_U32 detail_size;  // set to sizeof(TBRW_EVENT_DETAIL).
    TBRW_U16 detail_access_size; // size of the access (in bits) i.e. 8, 16,
32, 64
    TBRW_U8  detail_method;      // type of access – MSR, PCI, Memory, other
    TBRW_U8  detail_access;      // read or write

    TBRW_U64 detail_address;     // address of read/write
    TBRW_U64 detail_value;       // value of read/write
    TBRW_U8  legacy_command;
    TBRW_U8  reserved1;
    TBRW_U16 reserved2;
    TBRW_U32 reserved3;

} TBRW_EVENT_DETAIL;
```

8.11 Data Descriptor

A data descriptor is an array of TBRW_SAMPREC_DESC_ENTRY that fully describes the data for a corresponding data stream.

```
typedef struct __tbrw_samprec_desc_entry {
    union {
    TBRW_U64 force_8_byte_aligned;
        struct {
            TBRW_U32 desc_size;   // set to sizeof
(TBRW_SAMPREC_DESC_ENTRY). Used for versioning.
            TBRW_U32 desc_offset; // offset from start of sample record in
bytes
            TBRW_U16 desc_type;   // See ST_ types defined in
samprec_shared.h
            TBRW_U16 desc_subtype;// See SST_ types defined in
samprec_shared.h
            TBRW_U32 desc_data_size;     // in bytes
            TBRW_U64 desc_access_offset; // msr # or memory offset
            TBRW_U8  desc_access_type;   // read = 0, write = 1
            TBRW_U8  desc_access_method; // register = 0, memory = 1
            TBRW_U8  desc_sample_flag;   // internal driver sample flag
            TBRW_U8  reserved0;          // reserved
            TBRW_U32 reserved1;
            TBRW_STRING_OR_ID  desc_name; //human-readable name or comment
        };
    };
} TBRW_SAMPREC_DESC_ENTRY;
```

```
//Note that one usage model for filling in the data descriptor structure is
//to set desc_type = ST_NONE and set the desc_name to a string of your
choice.
//Then, when enumerating through the data descriptors you can parse the
desc_name
//to determine the type of this data descriptor.


typedef enum {
    ST_NONE = 0,
    ST_LEGACY_SAMPLE_RECORD,
    ST_IP,
    ST_PID,
    ST_TID,
    ST_PROCESSOR_NUMBER,
    ST_PROCESSOR_STATUS,
    ST_TIME_STAMP,
    ST_POWER,
    ST_INTERRUPT_FAULT_ADDR,
    ST_IEAR,
    ST_DEAR,
    ST_BRANCH_TRACE,
    ST_INST_TRACE,
    ST_PMD,
    ST_IEAR_PHYSICAL,
    ST_DEAR_PHYSICAL,
    ST_BRANCH_TRACE_PHYSICAL,
    ST_INST_TRACE_PHYSICAL,
    ST_PMD_PHYSICAL,
    ST_LEGACY_UNKNOWN, //unknown legacy type

    ST_SAMPLE_LAST,

    //
    // Extended sample record entries start here - these entries do
    // not physically reside in a sample record but can be computed
    // based on data in a physical sample record
    //
    // I.E. (bit 14 and up == 0) &&  (bit 13 == 1) && (bit 12 == 0)
    // means extended sample record entries
    // (decimal 8192 through 12287) = 4096 values
    //
    ST_START_EXT_SAMPLE_REC_ENTRIES = 0x2000,
    ST_END_EXT_SAMPLE_REC_ENTRIES = 0x2FFF,

    ST_PROCESS = 0x2000,                 // process name
    ST_PROCESS_PATH,             // process path
    ST_THREAD,                   // thread name
    ST_MODULE_SEGMENT_NAME,      // module segment name
    ST_SEG_NUM,                  // module segment number
    ST_SEG_OFFSET,
    ST_SEG_TYPE,
    ST_MODULE_LOAD_ADDRESS,      // module load address
    ST_MODULE,                   // module name
    ST_MODULE_PATH,              // module path
    ST_EVENT_ID,                 // event ID
    ST_EVENT,                    // event name
    ST_HARDWARE_THREAD,          // hardware thread
    ST_CORE,                     // core number
    ST_PACKAGE,                  // package number
    ST_SAMPLES,
    ST_EVENT_COUNTS,
    ST_FUNCTION,
    ST_FUNCTION_FULL_NAME,
    ST_CLASS,
    ST_FN_SIZE,
    ST_RVA,
    ST_FN_SEG_OFFSET,
    ST_FN_RVA,
    ST_WALL_CLOCK,               // Wall clock.


    ST_FNID_SUM,                      // e.g. sum(some_column)
    ST_FNID_AVG,                          // e.g. avg(some_column)
    ST_FNID_MIN,                          // e.g. min(some_column)
    ST_FNID_MAX,                          // e.g. max(some_column)


    // Bind related
    ST_PROCESS_IDX,
    ST_THREAD_IDX,
    ST_MODULE_IDX,
    ST_EVENT_IDX,
    ST_SAMPLE_IDX,

    ST_PERCENTAGE_TOTAL,
    ST_PERCENTAGE_SEL,

    ST_MODULE_IDX_FOR_PROCESS_NAME,
    ST_ANNOTATION,              // Annotation text.

    ST_POST_PROCESS_LAST,

    //
    // the VTune™ Analyzer ignores
    //
    // I.E. (bit 15 == 0) &&  (bit 14 == 1) means user-defined.
    // (decimal 16384 through 32767)
    //
    ST_START_USER_DEFINED = 0x4000,
    ST_END_USER_DEFINED = 0x7FFF,

    //
    // For future use...
    //
    // It is an error to use a type between
    // ST_RESERVED_START and ST_RESERVED_END, inclusive
    //
    ST_RESERVED_START = 0x8000,
    ST_RESERVED_END   = 0xffff

} TBRW_DESC_TYPE;

typedef enum {
    SST_NONE = 0,

    //
    // Indicates a no data should be filled in by the driver,
    // but is left empty for someone else to use
    //
    SST_BLANK_SPACE,
    // Subtypes for time
    SST_TS_MILLISECONDS,
    SST_TS_CPU_CYCLES,
    SST_TS_FSB_CYCLES,
    SST_TS_OTHER,
    SST_TS_SAMPLE_COUNT,
    SST_TS_NANOSECONDS,

    // subtypes for BTB register
    SST_LBR_TOS,
    SST_LBR_FROM,
    SST_LBR_TO,
    SST_LBR_FROM_TO,
    SST_LBR_OTHER,

    SST_IEAR_CONFIG,
    SST_IEAR_INST_ADDR,
    SST_IEAR_LATENCY,

    SST_DEAR_CONFIG,
    SST_DEAR_INST_ADDR,
    SST_DEAR_LATENCY,
    SST_DEAR_DATA_ADDR,

    SST_BTB_CONFIG,
    SST_BTB_INDEX,
    SST_BTB_EXTENSION,
    SST_BTB_DATA,

    SST_IPEAR_CONFIG,
    SST_IPEAR_INDEX,
    SST_IPEAR_EXTENSION,
    SST_IPEAR_DATA,

    SST_IIP,
    SST_IPSR,
    SST_EIP,
    SST_EFLAGS,
    SST_TSC,
    SST_ITC,
    SST_PSTATE,
    SST_IA32_PERF_STATUS,
    SST_IFA,

    //
    SST_RESERVED_START = 0x8000,
    SST_RESERVED_END = 0xffff

} TBRW_DESC_SUBTYPE;
```

NOTE: The data descriptor types and sub types listed above are defined in the `samprec_shared.h` header file.

8.12 Bind Structure

NOTE: one DATA_BIND_STRUCT exists per one sample record

```
typedef struct __data_bind_struct
{
    TBRW_U64 module_index; //index into the module array
    TBRW_U64 pid_index;    //index into the pid array
    TBRW_U64 tid_index;    //index into the tid array
} DATA_BIND_STRUCT;

//Note: there is a 1:1 mapping between elements of this type
//and the module record, i.e. one MODULE_BIND_STRUCT exists per module
typedef struct __module_bind_struct
{
    TBRW_U64 pid_index; //index into the pid array
} MODULE_BIND_STRUCT;

//Note: there is a 1:1 mapping between elements of this type
//and the tid record, i.e. one TID_BIND_STRUCT exists per tid
typedef struct __tid_bind_struct
{
    TBRW_U64 pid_index; //index into the pid array
} TID_BIND_STRUCT;
```

9 API Function Reference

For compatibility with the largest audience, the VTune analyzer reader/writer API's is defined in C. The implementation of the VTune analyzer reader/writer API itself is in either C or C++.

9.1 High Level Functions

TBRW_U32 TBRW_get_version(OUT TBRW_U32 *major, OUT TBRW_U32 *minor)

Gets a major and minor number representing the current version of the VTune analyzer reader/writer API.

TBRW_U32 TBRW_open(OUT TBRW_PTR *ptr, IN const TBRW_CHAR *filename, IN TBRW_U32 access_mode)

Open the VTune analyzer file.

Returns an opaque type passed to all the rest of the routines so the API can keep data per open (similar in concept to the fd passed back by a generic `open()`call). The access mode can be a combination of the file permissions flags defined in `tbrw_types.h`

```
TBRW_FILE_READ
TBRW_FILE_WRITE
TBRW_FILE_CREATE_ALWAYS
```

If possible, it is best to give read and write permissions, since doing so improves performance on subsequent accesses to the file.

NOTE: When you create new tb5 files, you must provide the `TBRW_FILE_CREATE_ALWAYS` flag.

TBRW_U32 TBRW_close (IN TBRW_PTR ptr)

Close the VTune analyzer file.

TBRW_U32 TBRW_error_string(IN TBRW_U32 error_code, OUT const TBRW_CHAR **error_string)

Convert a TBRW_U32 error code

This function is returned when any of the API calls are used into a string by calling this API. TBRW_U32 error codes are defined in the file tbrw.h.

You cannot modify the string (it is a const). If you wish to modify the string, make a copy and modify the copy.

TBRW_U32 TBRW_abort_cleanup_and_close(IN TBRW_PTR ptr)

Abort the use of the VTune analyzer file.

Use this function during abnormal error conditions. This function enables the API to do internal cleanup as required. For example, removing temporary files, or freeing internal memory.

TBRW_U32 TBRW_verify (IN TBRW_PTR ptr)

Verify that the currently opened file is a valid VTune analyzer file.

Use this routine to verify the file is a proper VTune analyzer file before trying to access the data.

TBRW_U32 TBRW_convert_uniqueid_to_string(IN TBRW_PTR ptr, IN TBRW_U32 size_of_buffer, IN TBRW_STRING_OR_ID *string_id, OUT TBRW_CHAR *buffer, OPTIONAL OUT TBRW_U32 *size_buffer_needed)

Returns a string corresponding to a string ID

When passed a unique string id, returns the string that corresponded to that unique id.

All strings in TBRW are represented by the TBRW_STRING_OR_ID data structure. This is the data structure:

```
typedef struct __string_or_id {  // on writes, it is a string pointer
                                 // on reads, it is a unique id.
    union {
        TBRW_U64   soi_uniqueid;// on reads, it is a unique id.
        TBRW_CHAR *soi_ptr;     // name string
    };
} TBRW_STRING_OR_ID;
```

The usage model for strings is as follows:

When writing a field of type `TBRW_STRING_OR_ID`, fill in the value for `soi_ptr`, which is a pointer to `wchar_t`. Do not worry about unique id's when writing.

When reading a field of type `TBRW_STRING_OR_ID` from a VTune analyzer file, the field contains the unique id of a string, represented by `soi_uniqueid`, and not the actual string itself. To get the actual string, call `TBRW_convert_uniqueid_to_string()` is needed. `TBRW_convert_uniqueid_to_string()` translates the `soi_uniqueid` to a `wchar_t` string, pointed to by `soi_ptr`. The `soi_ptr` returned should not be copied or stored, it's valid only until the next `TBRW_convert_uniqueid_to_string()` is called.

To save the string, you need to make a local copy of `soi_ptr`. If you have multiple threads calling this function, make sure to put appropriate synchronization primitives in place to make sure that one thread is done with the provided pointer (not just the call, but the use of the returned pointer) before another thread makes a call to this routine.

9.1.1 Global Section Management

TBRW_U32 TBRW_reading_section(IN TBRW_PTR ptr, IN TBRW_SECTION_IDENTIFIER section)

Tells the API you are going to be using section for reading.

TBRW_U32 TBRW_writing_section(IN TBRW_PTR ptr, IN TBRW_SECTION_IDENTIFIER section)

Tells the API you will use the section for writing.

If the section already exists, this call results in an error.

TBRW_U32 TBRW_done_section(IN TBRW_PTR ptr, IN TBRW_SECTION_IDENTIFIER section)

Tells the API you are done using section.

If writing, this call also does limited validation of the data local to the section. Every `TBRW_reading_section()` and `TBRW_writing_section()` must have a corresponding `TBRW_done_section()`.

9.1.2 Data Stream Management

TBRW_U32 TBRW_get_number_data_streams(IN TBRW_PTR ptr, OUT TBRW_U32 *numStreams)

TBRW_U32 TBRW_reading_stream(IN TBRW_PTR ptr, IN TBRW_U32 stream)

Tells the API you are going to be using the data stream for reading.

TBRW_U32 TBRW_writing_stream(IN TBRW_PTR ptr, IN TBRW_U32 stream)

Tells the API you will use the data stream for writing

This function also sets the comment and type of the stream. If the data stream already exists, this call results in an error.

TBRW_U32 TBRW_done_stream(IN TBRW_PTR ptr, IN TBRW_U32 stream)

Tells the API you are done using the data stream.

If writing, this call also does limited validation of the data stream. Every `TBRW_reading_stream(IN TBRW_U32 stream)` and `TBRW_writing_stream(IN TBRW_U32 stream)` must have a corresponding `TBRW_done_stream(IN TBRW_U32 stream)`.

9.1.3 Data Stream Section Management

TBRW_U32 TBRW_reading_stream_section(IN TBRW_PTR ptr, IN TBRW_U32 stream, TBRW_STREAM_SECTION_IDENTIFIER section)

Tells the API you will use a section of IN TBRW_U32 stream for reading.

TBRW_U32 TBRW_writing_stream_section(IN TBRW_PTR ptr, IN TBRW_U32 stream, TBRW_STREAM_SECTION_IDENTIFIER section)

Tells the API you will use a section of IN TBRW_U32 stream for writing

This function also sets the comment and type of the stream. If the section of `IN TBRW_U32` stream already exists, this call results in an error.

TBRW_U32 TBRW_done_stream_section(IN TBRW_PTR ptr, IN TBRW_U32 stream, TBRW_STREAM_SECTION_IDENTIFIER section)

Tells the API you are done using section of IN TBRW_U32 stream.

If writing, this call also does limited validation of the section of IN TBRW_U32 stream. Every `TBRW_reading_stream_section(IN TBRW_U32 stream, section)` and `TBRW_writing_stream_section(IN TBRW_U32 stream, section)` must have a corresponding `TBRW_done_stream_section(IN TBRW_U32 stream, section)`.

9.2 Global Section Access

9.2.1 Hardware section

TBRW_U32 TBRW_ set_system(IN TBRW_PTR ptr, IN TBRW_SYSTEM *system)

Write information about the entire system.

TBRW_U32 TBRW_ get_system(IN TBRW_PTR ptr, IN TBRW_U32 size_of_buffer, OUT TBRW_SYSTEM *buf_ptr, OPTIONAL OUT TBRW_U32 *size_buffer_used)

Read information about the entire system.

9.2.2 Software Section

TBRW_U32 TBRW_set_host(IN TBRW_PTR ptr, IN TBRW_HOST *host)

Set information about the host in the software section

TBRW_U32 TBRW_get_host(IN TBRW_PTR ptr, IN TBRW_U32 size_of_buffer, IN void *buf_ptr, OPTIONAL OUT TBRW_U32 *size_buffer_used)

Get information about the host from the software section

TBRW_U32 TBRW_set_os(IN TBRW_PTR ptr, IN TBRW_OS *os)

Set information about the OS in the software section

TBRW_U32 TBRW_get_os(IN TBRW_PTR ptr, IN TBRW_U32 size_of_buffer, IN void *buf_ptr, OPTIONAL OUT TBRW_U32 *size_buffer_used)

Get information about the OS from the software section

TBRW_U32 TBRW_set_application(IN TBRW_PTR ptr, IN TBRW_APPLICATION *application)

Set information about the application in the software section

TBRW_U32 TBRW_get_application(IN TBRW_PTR ptr, IN TBRW_U32 size_of_buffer, IN void *buf_ptr, OPTIONAL OUT TBRW_U32 *size_buffer_used)

Get information about the application from the software section

9.2.3 Process/Thread Section

TBRW_U32 TBRW_add_process(IN TBRW_PTR ptr, IN TBRW_PID *process)

Add information about a process to the process/thread section

TBRW_U32 TBRW_get_one_pid(IN TBRW_PTR ptr, IN TBRW_U64 pid_index, OUT const TBRW_PID **p_pid)

Get one PID pointer, given a PID index.

The pointer is valid until the next call to `BIND_get_one_pid` is made.

TBRW_U32 TBRW_enumerate_processes(IN TBRW_PTR ptr, IN TBRW_PID_CALLBACK *callback_func, IN void *user_ptr, IN TBRW_U64 start_index)

Get information about the processes from the process/thread section.

The `start_index` parameter indicates which process index to start enumerating from.

TBRW_U32 TBRW_add_thread(IN TBRW_PTR ptr, IN TBRW_TID *thread)

Add information about a thread to the process/thread section

TBRW_U32 TBRW_get_one_tid(IN TBRW_PTR ptr, IN TBRW_U64 tid_index, OUT const TBRW_TID **p_tid)

Get one tid pointer, given a tid index.

The pointer is valid until the next call to `TBRW_get_one_tid` is made.

TBRW_U32 TBRW_enumerate_threads(IN TBRW_PTR ptr, IN TBRW_TID_CALLBACK *callback_func, IN void *user_ptr, IN TBRW_U64 start_index)

Get information about the threads from the process/thread section.

The start_index parameter indicates which thread index to start enumerating from.

TBRW_U32 TBRW_bind_enumerate_threads(IN TBRW_PTR ptr, IN BIND_TID_CALLBACK *callback_func, IN void *user_ptr, IN TBRW_U64 start_index)

Get information about the TIDs and associated PID indexes.

The start_index parameter indicates which tid index to start enumerating from. `User_ptr` is passed through to the callback function untouched.

TBRW_U32 TBRW_get_size_of_tid_bind_entry(IN TBRW_PTR ptr, IN TBRW_U32 data_stream, OUT TBRW_U32 *sizeof_tid_bind_entry)

Get the size of a single entry of the tid bind structure.

This function gets the size of a single entry of the tid bind structure, for a particular data stream. You need to know the size of each entry in order to iterate through the tid bind structure. There is no need to call `TBRW_reading/done_stream()` before/after calling this function.

9.2.4 Module Section

TBRW_U32 TBRW_add_module(IN TBRW_PTR ptr, IN TBRW_MODULE *module)

Add information about a module to the module section

TBRW_U32 TBRW_get_one_module(IN TBRW_PTR ptr, IN TBRW_U64 module_index, OUT const TBRW_MODULE **p_module)

Get one module pointer, given a module index.

The pointer is valid until the next call to TBRW_get_one_module is made.

TBRW_U32 TBRW_enumerate_modules(IN TBRW_PTR ptr, IN TBRW_MODULE_CALLBACK *callback_func, IN void *user_ptr, IN TBRW_U64 start_index)

Get information about the modules from the module section.

The `start_index` parameter indicates which thread index to start enumerating from. `User_ptr` is passed through to the callback function untouched.

TBRW_U32 TBRW_bind_enumerate_modules(IN TBRW_PTR ptr, IN BIND_MODULE_CALLBACK *callback_func, IN void *user_ptr, IN TBRW_U64 start_index)

Get information about the modules and associated PID index and PID name index.

The `start_index` parameter indicates which module index to start enumerating from. `User_ptr` is passed through to the callback function untouched.

TBRW_U32 TBRW_get_size_of_module_bind_entry(IN TBRW_PTR ptr, IN TBRW_U32 data_stream, OUT TBRW_U32 *sizeof_module_bind_entry)

For a particular data stream, get the size of a single entry of the module bind structure.

You need to know the size of each entry in order to iterate through the module bind structure. There is no need to call `TBRW_reading/done_stream()` before/after calling this function.

9.2.5 Version Information Global Section

TBRW_U32 TBRW_set_version_info(IN TBRW_PTR tbrw_ptr, IN TBRW_VERSION_INFO *version_info)

Write the version information global data

TBRW_U32 TBRW_get_version_info(IN TBRW_PTR tbrw_ptr, IN TBRW_U32 size_of_buffer, IN void *buf_ptr, OPTIONAL OUT TBRW_U32 *size_buffer_used)

Get the version information global data

9.2.6 User-defined Global Section

TBRW_U32 TBRW_set_user_defined_global(IN TBRW_PTR ptr, IN TBRW_U32 size_of_data, IN void *data_ptr)

Write user-defined global data.

TBRW_U32 TBRW_get_user_defined_global(IN TBRW_PTR ptr, IN TBRW_U32 size_of_buffer, IN void *buf_ptr, OPTIONAL OUT TBRW_U32 *size_buffer_used)

Read user-defined global data.

9.3 Stream Section Access

9.3.1 Stream Information Section

TBRW_U32 TBRW_get_stream_info(IN TBRW_PTR tbrw_ptr, IN TBRW_U32 stream, IN TBRW_U32 size_of_buffer, IN void *buf_ptr, OPTIONAL OUT TBRW_U32 *size_buffer_used)

Get stream information data

TBRW_U32 TBRW_set_stream_info(IN TBRW_PTR tbrw_ptr, IN TBRW_U32 stream, IN TBRW_STREAM_INFO *stream_info)

Set stream information data.

9.3.2 Event Description Section

TBRW_U32 TBRW_add_event(IN TBRW_PTR ptr, IN TBRW_U32 stream, IN TBRW_EVENT *event_descriptor_entry)

Append a new event descriptor entry to the event descriptor.

TBRW_U32 TBRW_enumerate_events(IN TBRW_PTR ptr, IN TBRW_U32 stream, IN TBRW_EVENT_CALLBACK *callback_func, IN void *user_ptr, IN TBRW_U64 start_index)

Enumerates the event descriptor entries in the event descriptor.

The start_index parameter indicates which event index to start enumerating from.

9.3.3 Data Description Section

TBRW_U32 TBRW_add_data_descriptor_entry(IN TBRW_PTR ptr, IN TBRW_U32 stream, IN TBRW_SAMPREC_DESC_ENTRY *data_descriptor_entry)

Appends a new data descriptor entry to the data descriptor.

TBRW_U32 TBRW_enumerate_data_descriptor_entries(IN TBRW_PTR ptr, IN TBRW_U32 stream, TBRW_DATA_DESC_CALLBACK *callback_func, void *user_ptr)

Enumerates the data descriptor entries in the data descriptor.

9.3.4 Data Section

TBRW_U32 TBRW_add_data(IN TBRW_PTR ptr, IN TBRW_U32 stream, IN TBRW_U32 size_of_data_entry, IN void *data_entry)

Appends a data entry to the data section.

The data entry should be in the format described by the data description section.

TBRW_U32 TBRW_add_data_from_file(IN TBRW_PTR ptr, IN TBRW_U32 stream, TBRW_CHAR *filename)

Appends data entries in a binary file to the data section.

The data entries should be in the format described by the data description section.

TBRW_U32 TBRW_enumerate_data(IN TBRW_PTR ptr, IN TBRW_U32 stream, IN TBRW_DATA_CALLBACK *callback_func, IN void *user_ptr, IN TBRW_U64 start_index)

Gets data entries from the data section.

The data returned is in the format described by the data description section. The `start_index` parameter indicates which data entry index to start enumerating from.

TBRW_U32 TBRW_bind_enumerate_data(IN TBRW_PTR ptr, IN TBRW_U32 data_stream, IN BIND_DATA_CALLBACK *callback_func, IN void *user_ptr, IN TBRW_U64 start_index,)

For a particular data stream, get information about the sampling data and associated modules, PIDs, TIDs.

The `start_index` parameter indicates which data index to start enumerating from. `User_ptr` is passed through to the callback function untouched.

TBRW_U32 TBRW_get_size_of_data_bind_entry(IN TBRW_PTR ptr, IN TBRW_U32 data_stream, OUT TBRW_U32 *sizeof_data_bind_entry)

For a particular data stream, get the size of a single entry of the data bind structure.

You need to know the size of each entry in order to iterate through the data bind structure. There is no need to call `TBRW_reading/done_stream()` before/after calling this function.

TBRW_U32 TBRW_is_bound(IN TBRW_PTR ptr, IN TBRW_U32 data_stream OUT TBRW_U32 *is_bound)

Check if a particular data stream in the file is bound or not.

`Is_bound` is set to 1 if it is bound, 0 otherwise. There is no need to call `TBRW_reading/done_stream()` before/after calling this function.

TBRW_U32 TBRW_dobind(IN TBRW_PTR ptr, IN TBRW_U32 data_stream)

Do the binding for a particular data stream.

There is no need to call `TBRW_writing/done_stream()` before/after calling this function.

TBRW_U32 TBRW_unbind(IN TBRW_PTR ptr, IN TBRW_U32 data_stream)

Do unbind for a particular data stream.

There is no need to call `TBRW_writing/done_stream()` before/after calling this function. This function is currently not implemented.

9.3.5 User-defined stream section

TBRW_U32 TBRW_set_user_defined_stream(IN TBRW_PTR ptr, IN TBRW_U32 stream, IN TBRW_U32 size_of_data, IN void *data_ptr)

Write user-defined data to be stored with a stream.

TBRW_U32 TBRW_get_user_defined_stream(IN TBRW_PTR ptr, IN TBRW_U32 stream, IN TBRW_U32 size_of_buffer, IN void *buf_ptr, OPTIONAL OUT TBRW_U32 *size_buffer_used)

Read user-defined data stored with a stream.

9.4 String Conversion Utility Functions

The following are the utility functions that can be used to convert strings from utf8 format to wide char format and vice versa.

TBRW_U32 TBRW_convert_utf8_to_wcs (IN const char *utf8, OUT wchar_t *wcs, INOUT TBRW_U32 *wcs_size);

Convert a UTF-8-encoded string into a native wchar_t string.

A common usage is to call this function with `*wcs_size = 0`. This basically acts as a query, and wcs can be NULL. As long the conversion still succeeds internally, the return value is `TBRW_BUFFER_TOO_SMALL`, and `*wcs_size` is set to the number of characters needed in the output buffer. You can then allocate your buffer accordingly and call this API again.

TBRW_U32 TBRW_convert_wcs_to_utf8 (IN const wchar_t *wcs, OUT char *utf8, INOUT TBRW_U32 *utf8_size);

Convert a native wchar_t string into a UTF-8-encoded string.

A common usage is to call this function with `*utf8_size = 0`. This basically acts as a query, and utf8 can be NULL. As long the conversion still succeeds internally, the return value is `TBRW_BUFFER_TOO_SMALL`, and `*utf8_size` is set to the number of characters needed in the output buffer. You can then allocate your buffer accordingly and call this again.

9.5 Callback Functions

You need to provide call back function pointers to be able to enumerate data from various sections of the tb5 file as discussed in the previously. This section lists the callback functions and their purpose. The callback functions are declared in `tbrw_types.h` header file.

TBRW_U32 (*TBRW_DATA_CALLBACK)(void *data, TBRW_U32 data_size, TBRW_U32 num_entries, void *user_ptr);

Enumerates the data from the data stream section.

The data is returned in the void* data parameter along with data size and number of data entries.

TBRW_U32 (*TBRW_PID_CALLBACK)(TBRW_PID *pid, TBRW_U32 pid_data_size, TBRW_U32 num_entries, void *user_ptr);

Retrieves the processes information from the tb5 data file.

This function also retrieves the process data size and number of process entries.

TBRW_U32 (*TBRW_TID_CALLBACK)(TBRW_TID *tid, TBRW_U32 tid_data_size, TBRW_U32 num_entries, void *user_ptr);

Retrieves the thread related information from the tb5 data file.

This function also retrieves the thread data size and number of thread entries.

TBRW_U32 (*TBRW_MODULE_CALLBACK)(TBRW_MODULE *module, TBRW_U32 module_data_size, TBRW_U32 num_entries, void *user_ptr);

Retrieves the module information from the tb5 data file along with module data size and number of modules.

TBRW_U32 (*TBRW_EVENT_CALLBACK)(TBRW_EVENT *event, TBRW_U32 event_data_size, TBRW_U32 num_entries, void *user_ptr);

Retrieves the event information from the tb5 data file along with event data size and number of events used for collecting the data.

TBRW_U32 (*TBRW_DATA_DESC_CALLBACK)(TBRW_SAMPREC_DESC_ENTRY *data_desc, TBRW_U32 data_desc_size, TBRW_U32 num_entries, void *user_ptr);

Retrieves the data descriptor entry information from the tb5 data file along with data descriptor size and number of descriptors.

10 Usage Example

10.1 Writing a Stream Section

The following pseudo code writes the stream section to the tb5 file. See the TBRW examples in the VTune analyzer installation package for more details.

```
int main(int argc, char *argv[])
{
    TBRW_U32 ret_val;
    void *tbrw_ptr;
    int stream = 0;
    wchar_t *err_text = NULL;
    TBRW_STREAM_INFO stream_info;
    //fill in stream_info

    ret_val = TBRW_open(&tbrw_ptr, file_name, access_mode);
    if (ret_val != VT_SUCCESS) {
        printf(“TBRW_open failed\n”);
        return 1;
    }

    ret_val = TBRW_writing_stream(tbrw_ptr, stream);
    if (ret_val != VT_SUCCESS)
    {
        ret_val = TBRW_error_string(ret_val, &err_text);
        printf("TBRW_writing_stream number %d returned error \"%ls\"\n",
stream, err_text);
        ret_val = TBRW_abort_cleanup_and_close(tbrw_ptr);
        return 1;
    }

    ret_val = TBRW_writing_stream_section(tbrw_ptr, stream,
TBRW_STREAM_INFO_SECTION);
    //if error, handle as above
    ret_val = TBRW_set_stream_info(tbrw_ptr, stream, &stream_info);
    //if error, handle as above
    ret_val = TBRW_done_stream_section(tbrw_ptr, stream,
TBRW_STREAM_INFO_SECTION);
    //if error, handle as above

    ret_val = TBRW_done_stream(tbrw_ptr, stream);
    //if error, handle as above

    ret_val = TBRW_close(tbrw_ptr);
    if (ret_val != VT_SUCCESS)
    {
        printf("TBRW_close failed\n");
    }
    return 0;
}//end main
```